Sharding FAQ

2017 Dec 31 See all posts

Currently, in all blockchain protocols each node stores the entire state (account balances, contract code and storage, etc.) and processes all transactions. This provides a large amount of security, but greatly limits scalability: a blockchain cannot process more transactions than a single node can. In large part because of this, Bitcoin is limited to ~3–7 transactions per second, Ethereum to 7–15, etc.

However, this poses a question: are there ways to create a new mechanism, where only a small subset of nodes verifies each transaction? As long as there are sufficiently many nodes verifying each transaction that the system is still highly secure, but a sufficiently small percentage of the total validator set so that the system can process many transactions in parallel, could we not split up transaction processing between smaller groups of nodes to greatly increase a blockchain's total throughput?

Contents

What are some trivial but flawed ways of solving the problem?

There are three main categories of "easy solutions". The first is to give up on scaling individual blockchains, and instead assume that applications will be split among many different chains. This greatly increases throughput, but at a cost of security: an N-factor increase in throughput using this method necessarily comes with an N-factor decrease in security, as a level of resources 1/N the size of the whole ecosystem will be sufficient to attack any individual chain. Hence, it is arguably non-viable for more than small values of N.

The second is to simply increase the block size limit. This can work and in some situations may well be the correct prescription, as block sizes may well be constrained more by politics than by realistic technical considerations. But regardless of one's beliefs about any individual case such an approach inevitably has its limits: if one goes too far, then nodes running on consumer hardware will drop out, the network will start to rely exclusively on a very small number of supercomputers running the blockchain, which can lead to great centralization risk.

The third is "merge mining", a technique where there are many chains, but all chains share the same mining power (or, in proof of stake systems, stake). Currently, Namecoin gets a large portion of its security from the Bitcoin blockchain by doing this. If all miners participate, this theoretically can increase throughput by a factor of N without compromising security. However, this also has the problem that it increases the computational and storage load on each miner by a factor of N, and so in fact this solution is simply a stealthy form of block size increase.

This sounds like there's some kind of scalability trilemma at play. What is this trilemma and can we break through it?

The trilemma claims that blockchain systems can only at most have two of the following three properties:

- **Decentralization** (defined as the system being able to run in a scenario where each participant only has access to O(c) resources, i.e. a regular laptop or small VPS)
- **Scalability** (defined as being able to process O(n) > O(c) transactions)
- **Security** (defined as being secure against attackers with up to O(n) resources)

In the rest of this document, we'll continue using **c** to refer to the size of computational resources (including computation, bandwidth and storage) available to each node, and **n** to refer to the size of the ecosystem in some abstract sense; we assume that transaction load, state size, and the market cap of a cryptocurrency are all proportional to **n**. The key challenge of scalability is finding a way to achieve all three at the base layer.

What are some moderately simple but only partial ways of solving the scalability problem?

Many sharding proposals (e.g. this early BFT sharding proposal from Loi Luu et al at NUS, more recent application of similar ideas in Zilliqa, as well as this Merklix tree[1] approach that has been

suggested for Bitcoin) attempt to either only shard transaction processing or only shard state, without touching the other[2]. These efforts can lead to some gains in efficiency, but they run into the fundamental problem that they only solve one of the two bottlenecks. We want to be able to process 10,000+ transactions per second without either forcing every node to be a supercomputer or forcing every node to store a terabyte of state data, and this requires a comprehensive solution where the workloads of state storage, transaction processing and even transaction downloading and re-broadcasting at are all spread out across nodes. Particularly, the P2P network needs to also be modified to ensure that not every node receives all information from every other node.

What about approaches that do not try to "shard" anything?

Bitcoin-NG can increase scalability somewhat by means of an alternative blockchain design that makes it much safer for the network if nodes are spending large portions of their CPU time verifying blocks. In simple PoW blockchains, there are high centralization risks and the safety of consensus is weakened if capacity is increased to the point where more than about 5% of nodes' CPU time is spent verifying blocks; Bitcoin-NG's design alleviates this problem. However, this can only increase the scalability of transaction capacity by a constant factor of perhaps 5-50x[3,4], and does not increase the scalability of state. That said, Bitcoin-NG-style approaches are not mutually exclusive with sharding, and the two can certainly be implemented at the same time.

Channel-based strategies (lightning network, Raiden, etc) can scale transaction capacity by a constant factor but cannot scale state storage, and also come with their own unique sets of tradeoffs and limitations particularly involving denial-of-service attacks. On-chain scaling via sharding (plus other techniques) and off-chain scaling via channels are arguably both necessary and complementary.

There exist approaches that use advanced cryptography, such as Mimblewimble and strategies based on ZK-SNARKs (eg. Coda), to solve one specific part of the scaling problem: initial full node synchronization. Instead of verifying the entire history from genesis, nodes could verify a cryptographic proof that the current state legitimately follows from the history. These approaches do solve a legitimate problem, but they are not a substitute for sharding, as they do not remove the need for nodes to download and verify very large amounts of data to stay on the chain in real time.

How does Plasma, state channels and other layer 2 technologies fit into the trilemma?

In the event of a large attack on Plasma subchains, all users of the Plasma subchains would need to withdraw back to the root chain. If Plasma has O(N) users, then this will require O(N) transactions, and so O(N / C) time to process all of the withdrawals. If withdrawal delays are fixed to some D (i.e. the naive implementation), then as soon as N > C * D, there will not be enough space in the blockchain to process all withdrawals in time, and so the system will be insecure; in this mode, Plasma should be viewed as increasing scalability only by a (possibly large) constant factor. If withdrawal delays are flexible, so they automatically extend if there are many withdrawals being made, then this means that as N increases further and further, the amount of time that an attacker can force everyone's funds to get locked up increases, and so the level of "security" of the system decreases further and further in a certain sense, as extended denial of access can be viewed as a security failure, albeit one milder than total loss of access. However, this is a different *direction* of tradeoff from other solutions, and arguably a much milder tradeoff, hence why Plasma subchains are nevertheless a large improvement on the status quo.

Note that there is one design that states that: "Given a malicious operator (the worst case), the system degrades to an on-chain token. A malicious operator cannot steal funds and cannot deprive people of their funds for any meaningful amount of time."—https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/3675. See also here for related information.

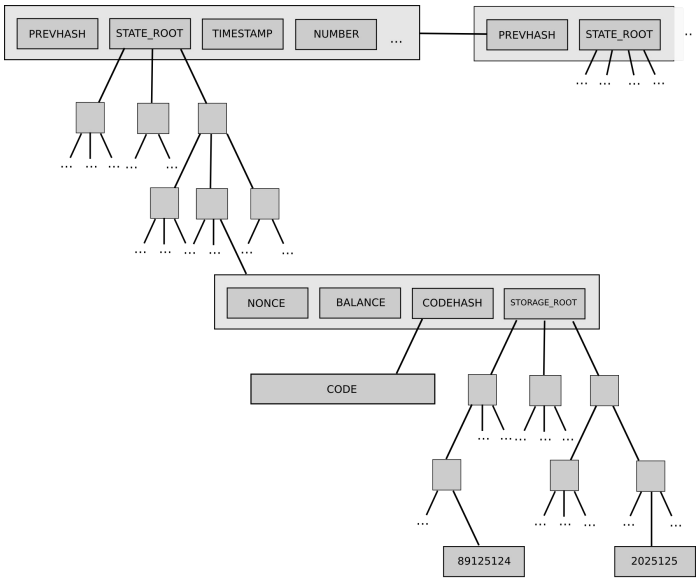
State channels have similar properties, though with different tradeoffs between versatility and speed of finality. Other layer 2 technologies include TrueBit off-chain interactive verification of execution and Raiden, which is another organisation working on state channels. Proof of stake with Casper (which is layer 1) would also improve scaling—it is more decentralizable, not requiring a computer that is able to mine, which tends towards centralized mining farms and institutionalized mining pools as difficulty increases and the size of the state of the blockchain increases.

Sharding is different to state channels and Plasma in that periodically notaries are pseudo-randomly assigned to vote on the validity of collations (analogous to blocks, but without an EVM state transition function in phase 1), then these collations are accepted into the main chain after the votes are verified by a committee on the main chain, via a sharding manager contract on the main chain. In phase 5 (see the roadmap for details), shards are tightly coupled to the main chain, so that if any shard or the main chain is invalid, the whole network is invalid. There are other differences between each mechanism, but at a high level, Plasma, state channels and Truebit are off-chain for an indefinite interval, connect to the main chain at the smart contract, layer 2 level, while they can draw back into and open up from the main chain, whereas shards are regularly linked to the main chain via consensus in-protocol.

See also these tweets from Vlad.

State size, history, cryptoeconomics, oh my! Define some of these terms before we move further!

- **State**: a set of information that represents the "current state" of a system; determining whether or not a transaction is valid, as well as the effect of a transaction, should in the simplest model depend only on state. Examples of state data include the UTXO set in bitcoin, balances + nonces + code + storage in ethereum, and domain name registry entries in Namecoin.
- **History**: an ordered list of all transactions that have taken place since genesis. In a simple model, the present state should be a deterministic function of the genesis state and the history.
- **Transaction**: an object that goes into the history. In practice, a transaction represents an operation that some user wants to make, and is cryptographically signed. In some systems transactions are called **blobs**, to emphasize the fact that in these systems these objects may contain arbitrary data and may not in all cases represent an attempt to perform some operation in the protocol.
- **State transition function**: a function that takes a state, applies a transaction and outputs a new state. The computation involved may involve adding and subtracting balances from accounts specified by the transaction, verifying digital signatures and running contract code.
- **Merkle tree**: a cryptographic hash tree structure that can store a very large amount of data, where authenticating each individual piece of data only takes O(log(n)) space and time. See here for details. In Ethereum, the transaction set of each block, as well as the state, is kept in a Merkle tree, where the roots of the trees are committed to in a block.
- **Receipt**: an object that represents an effect of a transaction that is not directly stored in the state, but which is still stored in a Merkle tree and committed to in a block header or in a special location in the state so that its existence can later be efficiently proven even to a node that does not have all of the data. Logs in Ethereum are receipts; in sharded models, receipts are used to facilitate asynchronous cross-shard communication.
- **Light client**: a way of interacting with a blockchain that only requires a very small amount (we'll say O(1), though O(log(c)) may also be accurate in some cases) of computational resources, keeping track of only the block headers of the chain by default and acquiring any needed information about transactions, state or receipts by asking for and verifying Merkle proofs of the relevant data on an as-needed basis.
- **State root**: the root hash of the Merkle tree representing the state[5]



The Ethereum 1.0 state tree, and how the state root fits into the block structure

What is the basic idea behind sharding?

We split the state and history up into K = O(n / c) partitions that we call "shards". For example, a sharding scheme on Ethereum might put all addresses starting with 0x00 into one shard, all addresses starting with 0x01 into another shard, etc. In the simplest form of sharding, each shard also has its own transaction history, and the effect of transactions in some shard k are limited to the state of shard k. One simple example would be a multi-asset blockchain, where there are K shards and each shard stores the balances and processes the transactions associated with one particular asset. In more advanced forms of sharding, some form of cross-shard communication capability, where transactions on one shard can trigger events on other shards, is also included.

What might a basic design of a sharded blockchain look like?

A simple approach is as follows. For simplicity, this design keeps track of data blobs only; it does not attempt to process a state transition function.

There exists a set of **validators** (ie. proof of stake nodes), who randomly get assigned the right to create **shard blocks**. During each **slot** (eg. an 8-second period of time), for each `k` in `[0...999]` a random validator gets selected, and given the right to create a block on "shard `k`", which might contain up to, say, 32 kb of data. Also, for each `k`, a set of 100 validators get selected as **attesters**. The header of a block together with at least 67 of the attesting signatures can be published as an object that gets included in the "main chain" (also called a **beacon chain**).

Note that there are now several "levels" of nodes that can exist in such a system:

- **Super-full node** - downloads the full data of the beacon chain and every shard block referenced in the beacon chain.
- **Top-level node** - processes the beacon chain blocks only, including the headers and signatures of the shard blocks, but does not download all the data of the shard blocks.
- **Single-shard node** - acts as a top-level node, but also fully downloads and verifies every collation on some specific shard that it cares more about.
- **Light node** - downloads and verifies the block headers of main chain blocks only; does not process any collation headers or transactions unless it needs to read some specific entry in the state of some specific shard, in which case it downloads the Merkle branch to the most recent collation header for that shard and from there downloads the Merkle proof of the desired value in the state.

What are the challenges here?

- **Single-shard takeover attacks** - what if an attacker takes over the majority of the validators responsible for attesting to one particular block, either to (respectively) prevent any collations from getting enough signatures or, worse, to submit collations that are invalid?
- **State transition execution** - single-shard takeover attacks are typically prevented with random sampling schemes, but such schemes also make it more difficult for validators to compute state roots, as they cannot have up-to-date state information for every shard that they could be assigned to. How do we ensure that light clients can still get accurate information about the state?
- **Fraud detection** - if an invalid collation or state claim does get made, how can nodes (including light nodes) be reliably informed of this so that they can detect the fraud and reject the collation if it is truly fraudulent?
- **Cross shard communication** - the above design supports no cross-shard communication. How do we add cross-shard communication safely?
- **The data availability problem** - as a subset of fraud detection, what about the specific case where data is missing from a collation?
- **Superquadratic sharding** - in the special case where n > c^2, in the simple design given above there would be more than O(c) collation headers, and so an ordinary node would not be able to process even just the top-level blocks. Hence, more than two levels of indirection between transactions and top-level block headers are required (i.e. we need "shards of shards"). What is the simplest and best way to do this?

However, the effect of a transaction may depend on *events that earlier took place in other shards*; a canonical example is transfer of money, where money can be moved from shard i to shard j by first creating a "debit" transaction that destroys coins in shard i, and then creating a "credit" transaction that creates coins in shard j, pointing to a receipt created by the debit transaction as proof that the credit is legitimate.

But doesn't the CAP theorem mean that fully secure distributed systems are impossible, and so sharding is futile?

The CAP theorem is a result that has to do with *distributed consensus*; a simple statement is: "in the cases that a network partition takes place, you have to choose either consistency or availability, you cannot have both". The intuitive argument is simple: if the network splits in half, and in one half I send a transaction "send my 10 coins to A" and in the other I send a transaction "send my 10 coins to B", then either the system is unavailable, as one or both transactions will not be processed, or it becomes inconsistent, as one half of the network will see the first transaction completed and the other half will see the second transaction completed. Note that the CAP theorem has nothing to do with scalability; it applies to any situation where multiple nodes need to agree on a value, regardless of the amount of data that they are agreeing on. All existing decentralized systems have found some compromise between availability and consistency; sharding does not make anything fundamentally harder in this respect.

What are the security models that we are operating under?

There are several competing models under which the safety of blockchain designs is evaluated:

- **Honest majority** (or honest supermajority): we assume that there is some set of validators and up to 50% (or 33% or 25%) of those validators are controlled by an attacker, and the remaining validators honestly follow the protocol. Honest majority models can have **non-adaptive** or **adaptive** adversaries; an adversary is adaptive if they can quickly choose which portion of the validator set to "corrupt", and non-adaptive if they can only make that choice far ahead of time. Note that the honest majority assumption may be higher for notary committees with a 61% honesty assumption.
- **Uncoordinated majority**: we assume that all validators are rational in a game-theoretic sense (except the attacker, who is motivated to make the network fail in some way), but no more than some fraction (often between 25% and 50%) are capable of coordinating their actions.
- **Coordinated choice**: we assume that most or all validators are controlled by the same actor, or are fully capable of coordinating on the economically optimal choice between themselves. We

can talk about the **cost to the coalition** (or profit to the coalition) of achieving some undesirable outcome.
- **Bribing attacker model**: we take the uncoordinated majority model, but instead of making the attacker be one of the participants, the attacker sits outside the protocol, and has the ability to bribe any participants to change their behavior. Attackers are modeled as having a **budget**, which is the maximum that they are willing to pay, and we can talk about their **cost**, the amount that they *end up paying* to disrupt the protocol equilibrium.

Bitcoin proof of work with Eyal and Sirer's selfish mining fix is robust up to 50% under the honest majority assumption, and up to ~23.21% under the uncoordinated majority assumption. Schellingcoin is robust up to 50% under the honest majority and uncoordinated majority assumptions, has ε (i.e. slightly more than zero) cost of attack in a coordinated choice model, and has a P + ε budget requirement and ε cost in a bribing attacker model due to P + epsilon attacks.

Hybrid models also exist; for example, even in the coordinated choice and bribing attacker models, it is common to make an **honest minority assumption** that some portion (perhaps 1-15%) of validators will act altruistically regardless of incentives. We can also talk about coalitions consisting of between 50-99% of validators either trying to disrupt the protocol or harm other validators; for example, in proof of work, a 51%-sized coalition can double its revenue by refusing to include blocks from all other miners.

The honest majority model is arguably highly unrealistic and has already been empirically disproven - see Bitcoin's SPV mining fork for a practical example. It proves too much: for example, an honest majority model would imply that honest miners are willing to voluntarily burn their own money if doing so punishes attackers in some way. The uncoordinated majority assumption may be realistic; there is also an intermediate model where the majority of nodes is honest but has a budget, so they shut down if they start to lose too much money.

The bribing attacker model has in some cases been criticized as being unrealistically adversarial, although its proponents argue that if a protocol is designed with the bribing attacker model in mind then it should be able to massively reduce the cost of consensus, as 51% attacks become an event that could be recovered from. We will evaluate sharding in the context of both uncoordinated majority and bribing attacker models. Bribing attacker models are similar to maximally-adaptive adversary models, except that the adversary has the additional power that it can solicit private information from all nodes; this distinction can be crucial, for example Algorand is secure under adaptive adversary models but not bribing attacker models because of how it relies on private information for random selection.

How can we solve the single-shard takeover attack in an uncoordinated majority model?

In short, random sampling. Each shard is assigned a certain number of notaries (e.g. 150), and the notaries that approve collations on each shard are taken from the sample for that shard. Samples can be reshuffled either semi-frequently (e.g. once every 12 hours) or maximally frequently (i.e. there is no real independent sampling process, notaries are randomly selected for each shard from a global pool every block).

Sampling can be explicit, as in protocols that choose specifically sized "committees" and ask them to vote on the validity and availability of specific collations, or it can be implicit, as in the case of "longest chain" protocols where nodes pseudorandomly assigned to build on specific collations and are expected to "windback verify" at least N ancestors of the collation they are building on.

The result is that even though only a few nodes are verifying and creating blocks on each shard at any given time, the level of security is in fact not much lower, in an honest or uncoordinated majority model, than what it would be if every single node was verifying and creating blocks. The reason is simple statistics: if you assume a ~67% honest supermajority on the global set, and if the size of the sample is 150, then with 99.999% probability the honest majority condition will be satisfied on the sample. If you assume a 75% honest supermajority on the global set, then that probability increases to 99.999999998% (see here for calculation details).

Hence, at least in the honest / uncoordinated majority setting, we have:

- **Decentralization** (each node stores only O(c) data, as it's a light client in O(c) shards and so stores O(1) * O(c) = O(c) data worth of block headers, as well as O(c) data corresponding to the recent history of one or several shards that it is assigned to at the present time)

- **Scalability** (with O(c) shards, each shard having O(c) capacity, the maximum capacity is n = O(c^2))
- **Security** (attackers need to control at least ~33% of the entire O(n)-sized validator pool in order to stand a chance of taking over the network).

In the bribing attacker model (or in the "very very adaptive adversary" model), things are not so easy, but we will get to this later. Note that because of the imperfections of sampling, the security threshold does decrease from 50% to ~30-40%, but this is still a surprisingly low loss of security for what may be a 100-1000x gain in scalability with no loss of decentralization.

How do you actually do this sampling in proof of work, and in proof of stake?

In proof of stake, it is easy. There already is an "active validator set" that is kept track of in the state, and one can simply sample from this set directly. Either an in-protocol algorithm runs and chooses 150 validators for each shard, or each validator independently runs an algorithm that uses a common source of randomness to (provably) determine which shard they are at any given time. Note that it is very important that the sampling assignment is "compulsory"; validators do not have a choice of what shard they go into. If validators could choose, then attackers with small total stake could concentrate their stake onto one shard and attack it, thereby eliminating the system's security.

In proof of work, it is more difficult, as with "direct" proof of work schemes one cannot prevent miners from applying their work to a given shard. It may be possible to use proof-of-file-access forms of proof of work to lock individual miners to individual shards, but it is hard to ensure that miners cannot quickly download or generate data that can be used for other shards and thus circumvent such a mechanism. The best known approach is through a technique invented by Dominic Williams called "puzzle towers", where miners first perform proof of work on a common chain, which then inducts them into a proof of stake-style validator pool, and the validator pool is then sampled just as in the proof-of-stake case.

One possible intermediate route might look as follows. Miners can spend a large (O(c)-sized) amount of work to create a new "cryptographic identity". The precise value of the proof of work solution then chooses which shard they have to make their next block on. They can then spend an O(1)-sized amount of work to create a block on that shard, and the value of that proof of work solution determines which shard they can work on next, and so on[8]. Note that all of these approaches make proof of work "stateful" in some way; the necessity of this is fundamental.

How is the randomness for random sampling generated?

First of all, it is important to note that even if random number generation is heavily exploitable, this is not a fatal flaw for the protocol; rather, it simply means that there is a medium to high centralization incentive. The reason is that because the randomness is picking fairly large samples, it is difficult to bias the randomness by more than a certain amount.

The simplest way to show this is through the binomial distribution, as described above; if one wishes to avoid a sample of size N being more than 50% corrupted by an attacker, and an attacker has p% of the global stake pool, the chance of the attacker being able to get such a majority during one round is:

$$\sum_{k=\frac{N}{2}}^{N} p^k (1-p)^{N-k} \binom{N}{k}$$

Here's a table for what this probability would look like in practice for various values of N and p:

<

	N = 50	N = 100	N = 150	N = 250
p = 0.4	0.0978	0.0271	0.0082	0.0009
p = 0.33	0.0108	0.0004	1.83 * 10-5	3.98 * 10-8
p = 0.25	0.0001	$6.63 * 10^{-8}$	$4.11 * 10^{-11}$	1.81 * 10-17

$p = 0.2 \quad 2.09 * 10^{-6}\ 2.14 * 10^{-11}\ 2.50 * 10^{-16}\ 3.96 * 10^{-26}$

Hence, for N >= 150, the chance that any given random seed will lead to a sample favoring the attacker is very small indeed[11,12]. What this means from the perspective of security of randomness is that the attacker needs to have a very large amount of freedom in choosing the random values order to break the sampling process outright. Most vulnerabilities in proof-of-stake randomness do not allow the attacker to simply choose a seed; at worst, they give the attacker many chances to select the most favorable seed out of many pseudorandomly generated options. If one is very worried about this, one can simply set N to a greater value, and add a moderately hard key-derivation function to the process of computing the randomness, so that it takes more than 2^{100} computational steps to find a way to bias the randomness sufficiently.

Now, let's look at the risk of attacks being made that try to influence the randomness more marginally, for purposes of profit rather than outright takeover. For example, suppose that there is an algorithm which pseudorandomly selects 1000 validators out of some very large set (each validator getting a reward of $1), an attacker has 10% of the stake so the attacker's average "honest" revenue 100, and at a cost of $1 the attacker can manipulate the randomness to "re-roll the dice" (and the attacker can do this an unlimited number of times).

Due to the central limit theorem, the standard deviation of the number of samples, and based on other known results in math the expected maximum of N random samples is slightly under M + S * sqrt(2 * log(N)) where M is the mean and S is the standard deviation. Hence the reward for manipulating the randomness and effectively re-rolling the dice (i.e. increasing N) drops off sharply, e.g. with 0 re-trials your expected reward is $100, with one re-trial it's $105.5, with two it's $108.5, with three it's $110.3, with four it's $111.6, with five it's $112.6 and with six it's $113.5. Hence, after five retrials it stops being worth it. As a result, an economically motivated attacker with ten percent of stake will (socially wastefully) spend $5 to get an additional revenue of $13, for a net surplus of $8.

However, this kind of logic assumes that one single round of re-rolling the dice is expensive. Many older proof of stake algorithms have a "stake grinding" vulnerability where re-rolling the dice simply means making a computation locally on one's computer; algorithms with this vulnerability are certainly unacceptable in a sharding context. Newer algorithms (see the "validator selection" section in the proof of stake FAQ) have the property that re-rolling the dice can only be done by voluntarily giving up one's spot in the block creation process, which entails giving up rewards and fees. The best way to mitigate the impact of marginal economically motivated attacks on sample selection is to find ways to increase this cost. One method to increase the cost by a factor of sqrt(N) from N rounds of voting is the majority-bit method devised by Iddo Bentov.

Another form of random number generation that is not exploitable by minority coalitions is the deterministic threshold signature approach most researched and advocated by Dominic Williams. The strategy here is to use a deterministic threshold signature to generate the random seed from which samples are selected. Deterministic threshold signatures have the property that the value is guaranteed to be the same regardless of which of a given set of participants provides their data to the algorithm, provided that at least ⅔ of participants do participate honestly. This approach is more obviously not economically exploitable and fully resistant to all forms of stake-grinding, but it has several weaknesses:

- **It relies on more complex cryptography** (specifically, elliptic curves and pairings). Other approaches rely on nothing but the random-oracle assumption for common hash algorithms.
- **It fails when many validators are offline**. A desired goal for public blockchains is to be able to survive very large portions of the network simultaneously disappearing, as long as a majority of the remaining nodes is honest; deterministic threshold signature schemes at this point cannot provide this property.
- **It's not secure in a bribing attacker or coordinated majority model** where more than 67% of validators are colluding. The other approaches described in the proof of stake FAQ above still make it expensive to manipulate the randomness, as data from all validators is mixed into the seed and making any manipulation requires either universal collusion or excluding other validators outright.

One might argue that the deterministic threshold signature approach works better in consistency-favoring contexts and other approaches work better in availability-favoring contexts.

What are the tradeoffs in making sampling more or less frequent?

Selection frequency affects just how adaptive adversaries can be for the protocol to still be secure against them; for example, if you believe that an adaptive attack (e.g. dishonest validators who discover that they are part of the same sample banding together and colluding) can happen in 6 hours but not less, then you would be okay with a sampling time of 4 hours but not 12 hours. This is an argument in favor of making sampling happen as quickly as possible.

The main challenge with sampling taking place every block is that reshuffling carries a very high amount of overhead. Specifically, verifying a block on a shard requires knowing the state of that shard, and so every time validators are reshuffled, validators need to download the entire state for the new shard(s) that they are in. This requires both a strong state size control policy (i.e. economically ensuring that the size of the state does not grow too large, whether by deleting old accounts, restricting the rate of creating new accounts or a combination of the two) and a fairly long reshuffling time to work well.

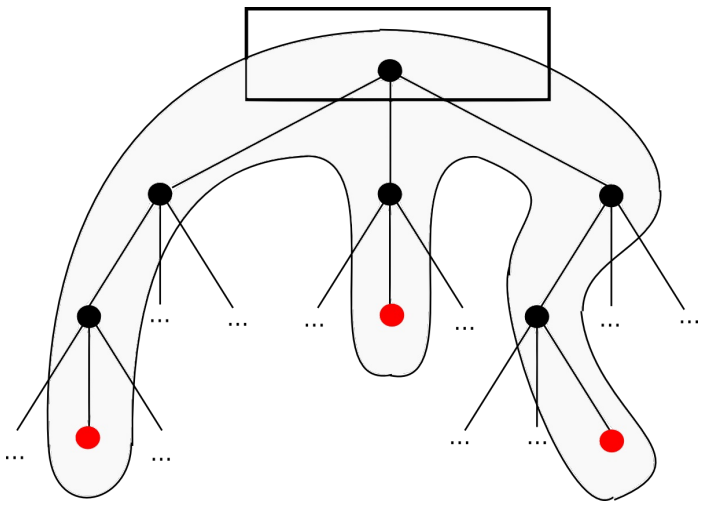
Currently, the Parity client can download and verify a full Ethereum state snapshot via "warp-sync" in ~2-8 hours, suggesting that reshuffling periods of a few days but not less are safe; perhaps this could be reduced somewhat by shrinking the state size via storage rent but even still reshuffling periods would need to be long, potentially making the system vulnerable to adaptive adversaries.

However, there are ways of completely avoiding the tradeoff, choosing the creator of the next collation in each shard with only a few minutes of warning but without adding impossibly high state downloading overhead. This is done by shifting responsibility for state storage, and possibly even state execution, away from collators entirely, and instead assigning the role to either users or an interactive verification protocol.

Can we force more of the state to be held user-side so that transactions can be validated without requiring validators to hold all state data?

See also: https://ethresear.ch/t/the-stateless-client-concept/172

The techniques here tend to involve requiring users to store state data and provide Merkle proofs along with every transaction that they send. A transaction would be sent along with a Merkle proof-of-correct-execution (or "witness"), and this proof would allow a node that only has the state root to calculate the new state root. This proof-of-correct-execution would consist of the subset of objects in the trie that would need to be traversed to access and verify the state information that the transaction must verify; because Merkle proofs are O(log(n)) sized, the proof for a transaction that accesses a constant number of objects would also be O(log(n)) sized.



The subset of objects in a Merkle tree that would need to be provided in a Merkle proof of a transaction that accesses several state objects

Implementing this scheme in its pure form has two flaws. First, it introduces O(log(n)) overhead (~10-30x in practice), although one could argue that this O(log(n)) overhead is not as bad as it seems

because it ensures that the validator can always simply keep state data in memory and thus it never needs to deal with the overhead of accessing the hard drive[9]. Second, it can easily be applied if the addresses that are accessed by a transaction are static, but is more difficult to apply if the addresses in question are dynamic - that is, if the transaction execution has code of the form `read(f(read(x)))` where the address of some state read depends on the execution result of some other state read. In this case, the address that the transaction sender thinks the transaction will be reading at the time that they send the transaction may well differ from the address that is actually read when the transaction is included in a block, and so the Merkle proof may be insufficient[10].

This can be solved with access lists (think: a list of accounts and subsets of storage tries), which specify statically what data transactions can access, so when a miner receives a transaction with a witness they can determine that the witness contains all of the data the transaction could possibly access or modify. However, this harms censorship resistance, making attacks similar in form to the attempted DAO soft fork possible.

Can we split data and execution so that we get the security from rapid shuffling data validation without the overhead of shuffling the nodes that perform state execution?

Yes. We can create a protocol where we split up validators into three roles with different incentives (so that the incentives do not overlap): **proposers or collators, a.k.a. prolators**, **notaries** and **executors**. Prollators are responsible for simply building a chain of collations; while notaries verify that the data in the collations is available. Prolators do not need to verify anything state-dependent (e.g. whether or not someone trying to send ETH has enough money). Executors take the chain of collations agreed to by the prolators as given, and then execute the transactions in the collations sequentially and compute the state. If any transaction included in a collation is invalid, executors simply skip over it. This way, validators that verify availability could be reshuffled instantly, and executors could stay on one shard.

There would be a light client protocol that allows light clients to determine what the state is based on claims signed by executors, but this protocol is NOT a simple majority-voting consensus. Rather, the protocol is an interactive game with some similarities to Truebit, where if there is great disagreement then light client simply execute specific collations or portions of collations themselves. Hence, light clients can get a correct view of the state even if 90% of the executors in the shard are corrupted, making it much safer to allow executors to be very infrequently reshuffled or even permanently shard-specific.

Choosing *what goes in* to a collation does require knowing the state of that collation, as that is the most practical way to know what will actually pay transaction fees, but this can be solved by further separating the role of collators (who agree on the history) and proposers (who propose individual collations) and creating a market between the two classes of actors; see here for more discussion on this. However, this approach has since been found to be flawed as per this analysis.

Can SNARKs and STARKs help?

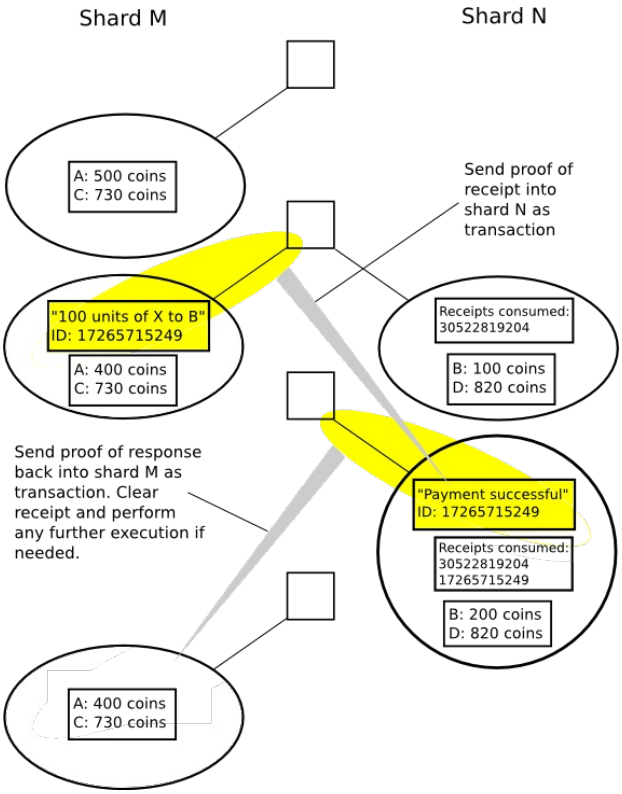
Yes! One can create a second-level protocol where a SNARK, STARK or similar succinct zero knowledge proof scheme is used to prove the state root of a shard chain, and proof creators can be rewarded for this. That said, shard chains to actually agree on what data gets included into the shard chains in the first place is still required.

How can we facilitate cross-shard communication?

The easiest scenario to satisfy is one where there are very many applications that individually do not have too many users, and which only very occasionally and loosely interact with each other; in this case, applications can live on separate shards and use cross-shard communication via receipts to talk to each other.

This typically involves breaking up each transaction into a "debit" and a "credit". For example, suppose that we have a transaction where account A on shard M wishes to send 100 coins to account B on shard N. The steps would looks as follows:

1. Send a transaction on shard M which (i) deducts the balance of A by 100 coins, and (ii) creates a receipt. A receipt is an object which is not saved in the state directly, but where the fact that the receipt was generated can be verified via a Merkle proof.
2. Wait for the first transaction to be included (sometimes waiting for finalization is required; this depends on the system).
3. Send a transaction on shard N which includes the Merkle proof of the receipt from (1). This transaction also checks in the state of shard N to make sure that this receipt is "unspent"; if it is, then it increases the balance of B by 100 coins, and saves in the state that the receipt is spent.
4. Optionally, the transaction in (3) also saves a receipt, which can then be used to perform further actions on shard M that are contingent on the original operation succeeding.



In more complex forms of sharding, transactions may in some cases have effects that spread out across several shards and may also synchronously ask for data from the state of multiple shards.

What is the train-and-hotel problem?

The following example is courtesy of Andrew Miller. Suppose that a user wants to purchase a train ticket and reserve a hotel, and wants to make sure that the operation is atomic - either both reservations succeed or neither do. If the train ticket and hotel booking applications are on the same shard, this is easy: create a transaction that attempts to make both reservations, and throws an exception and reverts everything unless both reservations succeed. If the two are on different shards, however, this is not so easy; even without cryptoeconomic / decentralization concerns, this is essentially the problem of atomic database transactions.

With asynchronous messages only, the simplest solution is to first reserve the train, then reserve the hotel, then once both reservations succeed confirm both; the reservation mechanism would prevent anyone else from reserving (or at least would ensure that enough spots are open to allow all reservations to be confirmed) for some period of time. However, this means that the mechanism relies on an extra security assumptions: that cross-shard messages from one shard can get included in another shard within some fixed period of time.

With cross-shard synchronous transactions, the problem is easier, but the challenge of creating a sharding solution capable of making cross-shard atomic synchronous transactions is itself decidedly nontrivial; see Vlad Zamfir's presentation which talks about merge blocks.

Another solution involves making contracts themselves movable across shards; see the proposed cross-shard locking scheme as well as this proposal where contracts can be "yanked" from one shard to another, allowing two contracts that normally reside on different shards to be temporarily moved to the same shard at which point a synchronous operation between them can happen.

What are the concerns about sharding through random sampling in a bribing attacker or coordinated choice model?

In a bribing attacker or coordinated choice model, the fact that validators are randomly sampled doesn't matter: whatever the sample is, either the attacker can bribe the great majority of the sample to do as the attacker pleases, or the attacker controls a majority of the sample directly and can direct the sample to perform arbitrary actions at low cost (O(c) cost, to be precise).

At that point, the attacker has the ability to conduct 51% attacks against that sample. The threat is further magnified because there is a risk of cross-shard contagion: if the attacker corrupts the state of a shard, the attacker can then start to send unlimited quantities of funds out to other shards and perform other cross-shard mischief. All in all, security in the bribing attacker or coordinated choice model is not much better than that of simply creating O(c) altcoins.

How can we improve on this?

In the context of state execution, we can use interactive verification protocols that are not randomly sampled majority votes, and that can give correct answers even if 90% of the participants are faulty; see Truebit for an example of how this can be done. For data availability, the problem is harder, though there are several strategies that can be used alongside majority votes to solve it.

What is the data availability problem, and how can we use erasure codes to solve it?

See https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding

Can we remove the need to solve data availability with some kind of fancy cryptographic accumulator scheme?

No. Suppose there is a scheme where there exists an object S representing the state (S could possibly be a hash) possibly as well as auxiliary information ("witnesses") held by individual users that can prove the presence of existing state objects (e.g. S is a Merkle root, the witnesses are the branches, though other constructions like RSA accumulators do exist). There exists an updating protocol where some data is broadcasted, and this data changes S to change the contents of the state, and also possibly changes witnesses.

Suppose some user has the witnesses for a set of N objects in the state, and M of the objects are updated. After receiving the update information, the user can check the new status of all N objects, and thereby see which M were updated. Hence, the update information itself encoded at least ~M * log(N) bits of information. Hence, the update information that everyone needs for receive to implement the effect of M transactions must necessarily be of size O(M). 14

So this means that we can actually create scalable sharded blockchains where the cost of making anything bad happen is

proportional to the size of the entire validator set?

There is one trivial attack by which an attacker can always burn O(c) capital to temporarily reduce the quality of a shard: spam it by sending transactions with high transaction fees, forcing legitimate users to outbid you to get in. This attack is unavoidable; you could compensate with flexible gas limits, and you could even try "transparent sharding" schemes that try to automatically re-allocate nodes to shards based on usage, but if some particular application is non-parallelizable, Amdahl's law guarantees that there is nothing you can do. The attack that is opened up here (reminder: it only works in the Zamfir model, not honest/uncoordinated majority) is arguably not substantially worse than the transaction spam attack. Hence, we've reached the known limit for the security of a single shard, and there is no value in trying to go further.

Let's walk back a bit. Do we actually need any of this complexity if we have instant shuffling? Doesn't instant shuffling basically mean that each shard directly pulls validators from the global validator pool so it operates just like a blockchain, and so sharding doesn't actually introduce any new complexities?

Kind of. First of all, it's worth noting that proof of work and simple proof of stake, even without sharding, both have very low security in a bribing attacker model; a block is only truly "finalized" in the economic sense after O(n) time (as if only a few blocks have passed, then the economic cost of replacing the chain is simply the cost of starting a double-spend from before the block in question). Casper solves this problem by adding its finality mechanism, so that the economic security margin immediately increases to the maximum. In a sharded chain, if we want economic finality then we need to come up with a chain of reasoning for why a validator would be willing to make a very strong claim on a chain based solely on a random sample, when the validator itself is convinced that the bribing attacker and coordinated choice models may be true and so the random sample could potentially be corrupted.

You mentioned transparent sharding. I'm 12 years old and what is this?

Basically, we do not expose the concept of "shards" directly to developers, and do not permanently assign state objects to specific shards. Instead, the protocol has an ongoing built-in load-balancing process that shifts objects around between shards. If a shard gets too big or consumes too much gas it can be split in half; if two shards get too small and talk to each other very often they can be combined together; if all shards get too small one shard can be deleted and its contents moved to various other shards, etc.

Imagine if Donald Trump realized that people travel between New York and London a lot, but there's an ocean in the way, so he could just take out his scissors, cut out the ocean, glue the US east coast and Western Europe together and put the Atlantic beside the South Pole - it's kind of like that.

What are some advantages and disadvantages of this?

- Developers no longer need to think about shards

- There's the possibility for shards to adjust manually to changes in gas prices, rather than relying on market mechanics to increase gas prices in some shards more than others
- There is no longer a notion of reliable co-placement: if two contracts are put into the same shard so that they can interact with each other, shard changes may well end up separating them
- More protocol complexity

The co-placement problem can be mitigated by introducing a notion of "sequential domains", where contracts may specify that they exist in the same sequential domain, in which case synchronous communication between them will always be possible. In this model a shard can be viewed as a set of sequential domains that are validated together, and where sequential domains can be rebalanced between shards if the protocol determines that it is efficient to do so.

How would synchronous cross-shard messages work?

The process becomes much easier if you view the transaction history as being already settled, and are simply trying to calculate the state transition function. There are several approaches; one fairly simple approach can be described as follows:

- A transaction may specify a set of shards that it can operate in
- In order for the transaction to be effective, it must be included at the same block height in all of these shards.
- Transactions within a block must be put in order of their hash (this ensures a canonical order of execution)

A client on shard X, if it sees a transaction with shards (X, Y), requests a Merkle proof from shard Y verifying (i) the presence of that transaction on shard Y, and (ii) what the pre-state on shard Y is for those bits of data that the transaction will need to access. It then executes the transaction and commits to the execution result. Note that this process may be highly inefficient if there are many transactions with many different "block pairings" in each block; for this reason, it may be optimal to simply require blocks to specify sister shards, and then calculation can be done more efficiently at a per-block level. This is the basis for how such a scheme could work; one could imagine more complex designs. However, when making a new design, it's always important to make sure that low-cost denial of service attacks cannot arbitrarily slow state calculation down.

What about semi-asynchronous messages?

Vlad Zamfir created a scheme by which asynchronous messages could still solve the "book a train and hotel" problem. This works as follows. The state keeps track of all operations that have been recently made, as well as the graph of which operations were triggered by any given operation (including cross-shard operations). If an operation is reverted, then a receipt is created which can then be used to revert any effect of that operation on other shards; those reverts may then trigger their own reverts and so forth. The argument is that if one biases the system so that revert messages can propagate twice as fast as other kinds of messages, then a complex cross-shard transaction that finishes executing in K rounds can be fully reverted in another K rounds.

The overhead that this scheme would introduce has arguably not been sufficiently studied; there may be worst-case scenarios that trigger quadratic execution vulnerabilities. It is clear that if transactions have effects that are more isolated from each other, the overhead of this mechanism is lower; perhaps isolated executions can be incentivized via favorable gas cost rules. All in all, this is one of the more promising research directions for advanced sharding.

What are guaranteed cross-shard calls?

One of the challenges in sharding is that when a call is made, there is by default no hard protocol-provided guarantee that any asynchronous operations created by that call will be made within any particular timeframe, or even made at all; rather, it is up to some party to send a transaction in the destination shard triggering the receipt. This is okay for many applications, but in some cases it may be problematic for several reasons:

- There may be no single party that is clearly incentivized to trigger a given receipt. If the sending of a transaction benefits many parties, then there could be **tragedy-of-the-commons effects**

where the parties try to wait longer until someone else sends the transaction (i.e. play "chicken"), or simply decide that sending the transaction is not worth the transaction fees for them individually.
- **Gas prices across shards may be volatile**, and in some cases performing the first half of an operation compels the user to "follow through" on it, but the user may have to end up following through at a much higher gas price. This may be exacerbated by DoS attacks and related forms of **griefing**.
- Some applications rely on there being an upper bound on the "latency" of cross-shard messages (e.g. the train-and-hotel example). Lacking hard guarantees, such applications would have to have **inefficiently large safety margins**.

One could try to come up with a system where asynchronous messages made in some shard automatically trigger effects in their destination shard after some number of blocks. However, this requires every client on each shard to actively inspect all other shards in the process of calculating the state transition function, which is arguably a source of inefficiency. The best known compromise approach is this: when a receipt from shard A at height `height_a` is included in shard B at height `height_b`, if the difference in block heights exceeds `MAX_HEIGHT`, then all validators in shard B that created blocks from `height_a + MAX_HEIGHT + 1` to `height_b - 1` are penalized, and this penalty increases exponentially. A portion of these penalties is given to the validator that finally includes the block as a reward. This keeps the state transition function simple, while still strongly incentivizing the correct behavior.

Wait, but what if an attacker sends a cross-shard call from every shard into shard X at the same time? Wouldn't it be mathematically impossible to include all of these calls in time?

Correct; this is a problem. Here is a proposed solution. In order to make a cross-shard call from shard A to shard B, the caller must pre-purchase "congealed shard B gas" (this is done via a transaction in shard B, and recorded in shard B). Congealed shard B gas has a fast demurrage rate: once ordered, it loses 1/k of its remaining potency every block. A transaction on shard A can then send the congealed shard B gas along with the receipt that it creates, and it can be used on shard B for free. Shard B blocks allocate extra gas space specifically for these kinds of transactions. Note that because of the demurrage rules, there can be at most GAS_LIMIT * k worth of congealed gas for a given shard available at any time, which can certainly be filled within k blocks (in fact, even faster due to demurrage, but we may need this slack space due to malicious validators). In case too many validators maliciously fail to include receipts, we can make the penalties fairer by exempting validators who fill up the "receipt space" of their blocks with as many receipts as possible, starting with the oldest ones.

Under this pre-purchase mechanism, a user that wants to perform a cross-shard operation would first pre-purchase gas for all shards that the operation would go into, over-purchasing to take into account the demurrage. If the operation would create a receipt that triggers an operation that consumes 100000 gas in shard B, the user would pre-buy 100000 * e (i.e. 271818) shard-B congealed gas. If that operation would in turn spend 100000 gas in shard C (i.e. two levels of indirection), the user would need to pre-buy 100000 * e^2 (i.e. 738906) shard-C congealed gas. Notice how once the purchases are confirmed, and the user starts the main operation, the user can be confident that they will be insulated from changes in the gas price market, unless validators voluntarily lose large quantities of money from receipt non-inclusion penalties.

Congealed gas? This sounds interesting for not just cross-shard operations, but also reliable intra-shard scheduling

Indeed; you could buy congealed shard A gas inside of shard A, and send a guaranteed cross-shard call from shard A to itself. Though note that this scheme would only support scheduling at very short time intervals, and the scheduling would not be exact to the block; it would only be guaranteed to

happen within some period of time.

Does guaranteed scheduling, both intra-shard and cross-shard, help against majority collusions trying to censor transactions?

Yes. If a user fails to get a transaction in because colluding validators are filtering the transaction and not accepting any blocks that include it, then the user could send a series of messages which trigger a chain of guaranteed scheduled messages, the last of which reconstructs the transaction inside of the EVM and executes it. Preventing such circumvention techniques is practically impossible without shutting down the guaranteed scheduling feature outright and greatly restricting the entire protocol, and so malicious validators would not be able to do it easily.

Could sharded blockchains do a better job of dealing with network partitions?

The schemes described in this document would offer no improvement over non-sharded blockchains; realistically, every shard would end up with some nodes on both sides of the partition. There have been calls (e.g. from IPFS's Juan Benet) for building scalable networks with the specific goal that networks can split up into shards as needed and thus continue operating as much as possible under network partition conditions, but there are nontrivial cryptoeconomic challenges in making this work well.

One major challenge is that if we want to have location-based sharding so that geographic network partitions minimally hinder intra-shard cohesion (with the side effect of having very low intra-shard latencies and hence very fast intra-shard block times), then we need to have a way for validators to choose which shards they are participating in. This is dangerous, because it allows for much larger classes of attacks in the honest/uncoordinated majority model, and hence cheaper attacks with higher griefing factors in the Zamfir model. Sharding for geographic partition safety and sharding via random sampling for efficiency are two fundamentally different things.

Second, more thinking would need to go into how applications are organized. A likely model in a sharded blockchain as described above is for each "app" to be on some shard (at least for small-scale apps); however, if we want the apps themselves to be partition-resistant, then it means that all apps would need to be cross-shard to some extent.

One possible route to solving this is to create a platform that offers both kinds of shards - some shards would be higher-security "global" shards that are randomly sampled, and other shards would be lower-security "local" shards that could have properties such as ultra-fast block times and cheaper transaction fees. Very low-security shards could even be used for data-publishing and messaging.

What are the unique challenges of pushing scaling past n = O(c^2)?

There are several considerations. First, the algorithm would need to be converted from a two-layer algorithm to a stackable n-layer algorithm; this is possible, but is complex. Second, n / c (i.e. the ratio between the total computation load of the network and the capacity of one node) is a value that happens to be close to two constants: first, if measured in blocks, a timespan of several hours, which is an acceptable "maximum security confirmation time", and second, the ratio between rewards and deposits (an early computation suggests a 32 ETH deposit size and a 0.05 ETH block reward for Casper). The latter has the consequence that if rewards and penalties on a shard are escalated to be on the scale of validator deposits, the cost of continuing an attack on a shard will be O(n) in size.

Going above c^2 would likely entail further weakening the kinds of security guarantees that a system can provide, and allowing attackers to attack individual shards in certain ways for extended periods of time at medium cost, although it should still be possible to prevent invalid state from being finalized and to prevent finalized state from being reverted unless attackers are willing to pay an O(n) cost. However, the rewards are large - a super-quadratically sharded blockchain could be used as a general-purpose tool for nearly all decentralized applications, and could sustain transaction fees

that makes its use virtually free.

What about heterogeneous sharding?

Abstracting the execution engine or allowing multiple execution engines to exist results in being able to have a different execution engine for each shard. Due to Casper CBC being able to explore the full tradeoff triangle, it is possible to alter the parameters of the consensus engine for each shard to be at any point of the triangle. However, CBC Casper has not been implemented yet, and heterogeneous sharding is nothing more than an idea at this stage; the specifics of how it would work has not been designed nor implemented. Some shards could be optimized to have fast finality and high throughput, which is important for applications such as EFTPOS transactions, while maybe most could have a moderate or reasonable amount each of finality, throughput and decentralization (number of validating nodes), and applications that are prone to a high fault rate and thus require high security, such as torrent networks, privacy focused email like Proton mail, etc., could optimize for a high decentralization, low finality and high throughput, etc. See also https://twitter.com/VladZamfir/status/932320997021171712 and https://ethresear.ch/t/heterogeneous-sharding/1979/2.

Footnotes

1. Merklix tree == Merkle Patricia tree

2. Later proposals from the NUS group do manage to shard state; they do this via the receipt and state-compacting techniques that I describe in later sections in this document. (This is Vitalik Buterin writing as the creator of this Wiki.)

3. There are reasons to be conservative here. Particularly, note that if an attacker comes up with worst-case transactions whose ratio between processing time and block space expenditure (bytes, gas, etc) is much higher than usual, then the system will experience very low performance, and so a safety factor is necessary to account for this possibility. In traditional blockchains, the fact that block processing only takes ~1-5% of block time has the primary role of protecting against centralization risk but serves double duty of protecting against denial of service risk. In the specific case of Bitcoin, its current worst-case known quadratic execution vulnerability arguably limits any scaling at present to ~5-10x, and in the case of Ethereum, while all known vulnerabilities are being or have been removed after the denial-of-service attacks, there is still a risk of further discrepancies particularly on a smaller scale. In Bitcoin NG, the need for the former is removed, but the need for the latter is still there.

4. A further reason to be cautious is that increased state size corresponds to reduced throughput, as nodes will find it harder and harder to keep state data in RAM and so need more and more disk accesses, and databases, which often have an O(log(n)) access time, will take longer and longer to access. This was an important lesson from the last Ethereum denial-of-service attack, which bloated the state by ~10 GB by creating empty accounts and thereby indirectly slowed processing down by forcing further state accesses to hit disk instead of RAM.

5. In sharded blockchains, there may not necessarily be in-lockstep consensus on a single global state, and so the protocol never asks nodes to try to compute a global state root; in fact, in the protocols presented in later sections, each shard has its own state, and for each shard there is a mechanism for committing to the state root for that shard, which represents that shard's state

6. #MEGA

7. If a non-scalable blockchain upgrades into a scalable blockchain, the author's recommended path is that the old chain's state should simply become a single shard in the new chain.

8. For this to be secure, some further conditions must be satisfied; particularly, the proof of work must be non-outsourceable in order to prevent the attacker from determining which *other miners' identities* are available for some given shard and mining on top of those.

9. Recent Ethereum denial-of-service attacks have proven that hard drive access is a primary bottleneck to blockchain scalability.

10. You could ask: well why don't validators fetch Merkle proofs just-in-time? Answer: because doing so is a ~100-1000ms roundtrip, and executing an entire complex transaction within that time could be prohibitive.

11. One hybrid solution that combines the normal-case efficiency of small samples with the greater robustness of larger samples is a multi-layered sampling scheme: have a consensus between 50 nodes that requires 80% agreement to move forward, and then only if that consensus fails to be reached then fall back to a 250-node sample. N = 50 with an 80% threshold has only a 8.92 * 10-9 failure rate even against attackers with p = 0.4, so this does not harm security at all under an honest or uncoordinated majority model.

12. The probabilities given are for one single shard; however, the random seed affects O(c) shards and the attacker could potentially take over any one of them. If we want to look at O(c) shards simultaneously, then there are two cases. First, if the grinding process is computationally bounded, then this fact does not change the calculus at all, as even though there are now O(c) chances of success per round, checking success takes O(c) times as much work. Second, if the grinding process is economically bounded, then this indeed calls for somewhat higher safety factors (increasing N by 10-20 should be sufficient) although it's important to note that the goal of an attacker in a profit-motivated manipulation attack is to increase their participation across all shards in any case, and so that is the case that we are already investigating.

13. See Parity's Polkadotpaper for further description of how their "fishermen" concept works. For up-to-date info and code for Polkadot, see here.

14. Thanks to Justin Drake for pointing me to cryptographic accumulators, as well as this paper that gives the argument for the impossibility of sublinear batching. See also this thread: https://ethresear.ch/t/accumulators-scalability-of-utxo-blockchains-and-data-availability/176

Further reading related to sharding, and more generally scalability and research, is available here and here.