Nightshade: Near Protocol Sharding Design

Alex Skidanov
/AlexSkidanov
alex@nearprotocol.com

Illia Polosukhin
/ilblackdragon
illia@nearprotocol.com

July 2019

Contents

Introduction

It is well-known that Ethereum, the most used general purpose blockchain at the time of this writing, can only process less than 20 transactions per second on the main chain. This limitation, coupled with the popularity of the network, leads to high gas prices (the cost of executing a transaction on the network) and long confirmation times; despite the fact that at the time of this writing a new block is produced approximately every 10–20 seconds the average time it actually takes for a transaction to be added to the blockchain is 1.2 minutes, according to ETH Gas Station. Low throughput, high prices, and high latency all make Ethereum not suitable to run services that need to scale with adoption.

The main reason for Ethereum low throughput is that every node in the network needs to process every single transaction. Developers have proposed many solutions to address the issue of throughput on the protocol level. These solutions can be mostly separated into those that delegate all the computation to a small set of powerful nodes, and those that have each node in the network only do a subset of the total amount of work. An example of the former approach is Solana that through careful low level optimizations and GPU usage can reach hundreds of thousand simple payment transactions per second processed by each node in the system. Algorand, SpaceMesh, Thunder all fit into the former category, building various improvements in the consensus and the structure of the blockchain itself to run more transactions than Ethereum, but still bounded by what a single (albeit very powerful) machine can process.

The latter approach, in which the work is split among all the participating nodes, is called sharding. This is how Ethereum Foundation currently plans to scale Ethereum. At the time of this writing the spec is not finalized yet, the most recent spec can be found here: https://github.com/ethereum/eth2.0-specs.

Near Protocol is also built on sharding. The Near team, which includes several ex-MemSQL engineers responsible for building sharding, cross-shard transactions and distributed JOINs, as well as five ex-Googlers, has significant industry expertise in building distributed systems.

This document outlines the general approach to blockchain sharding, the major problems that need to be overcome, including state validity and data availability problems, and presents Nightshade, the solution Near Protocol is built upon that addresses those issues.

1 Sharding Basics[1]

Let's start with the simplest approach to sharding. In this approach instead of running one blockchain, we will run multiple, and call each such blockchain a "shard". Each shard will have its own set of validators. Here and below we use a generic term "validator" to refer to participants that verify transactions and produce blocks, either by mining, such as in Proof of Work, or via a voting-based

[1] This section was previously published at https://near.ai/shard1. If you read it before, skip to the next section.

mechanism. For now let's assume that the shards never communicate with each other.

This design, though simple, is sufficient to outline some initial major challenges in sharding.

1.1 Validator partitioning and Beacon chains

Say that the system comprises 10 shards. The first challenge is that with each shard having its own validators, each shard is now 10 times less secure than the entire chain. So if a non-sharded chain with X validators decides to hard-fork into a sharded chain, and splits X validators across 10 shards, each shard now only has X/10 validators, and corrupting one shard only requires corrupting 5.1% (51% / 10) of the total number of validators (see figure 1),

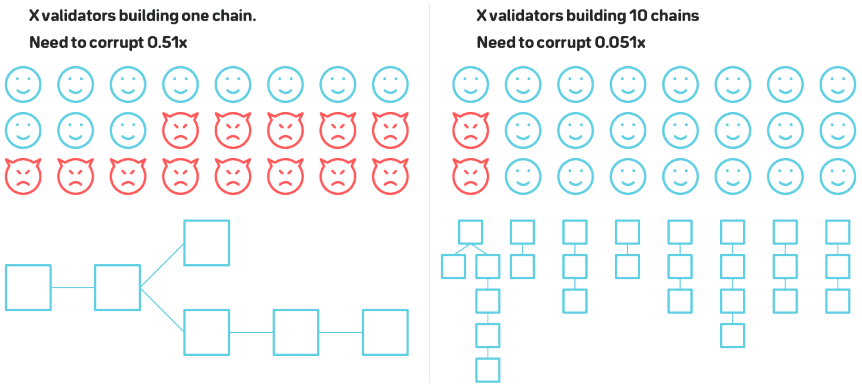

Figure 1: Splitting the validators across shards

which brings us to the second point: who chooses validators for each shard? Controlling 5.1% of validators is only damaging if all those 5.1% of validators are in the same shard. If validators can't choose which shard they get to validate in, a participant controlling 5.1% of the validators is highly unlikely to get all their validators in the same shard, heavily reducing their ability to compromise the system.

Almost all sharding designs today rely on some source of randomness to assign validators to shards. Randomness on blockchain on itself is a very challenging topic and is out of scope for this document. For now let's assume there's some source of randomness we can use. We will cover validators assignment in more detail in section 2.1.

Both randomness and validator assignment require computation that is not specific to any particular shard. For that computation, practically all existing designs have a separate blockchain that is tasked with performing operations necessary for the maintenance of the entire network. Besides generating random

numbers and assigning validators to the shards, these operations often also include receiving updates from shards and taking snapshots of them, processing stakes and slashing in Proof-of-Stake systems, and rebalancing shards when that feature is supported. Such chain is called a Beacon chain in Ethereum, a Relay chain in PolkaDot, and the Cosmos Hub in Cosmos.

Throughout this document we will refer to such chain as a Beacon chain. The existence of the Beacon chain brings us to the next interesting topic, the quadratic sharding.

1.2 Quadratic sharding

Sharding is often advertised as a solution that scales infinitely with the number of nodes participating in the network operation. While it is in theory possible to design such a sharding solution, any solution that has the concept of a Beacon chain doesn't have infinite scalability. To understand why, note that the Beacon chain has to do some bookkeeping computation, such as assigning validators to shards, or snapshotting shard chain blocks, that is proportional to the number of shards in the system. Since the Beacon chain is itself a single blockchain, with computation bounded by the computational capabilities of nodes operating it, the number of shards is naturally limited.

However, the structure of a sharded network does bestow a multiplicative effect on any improvements to its nodes. Consider the case in which an arbitrary improvement is made to the efficiency of nodes in the network which will allow them faster transaction processing times.

If the nodes operating the network, including the nodes in the Beacon chain, become four times faster, then each shard will be able to process four times more transactions, and the Beacon chain will be able to maintain 4 times more shards. The throughput across the system will increase by the factor of $4 \times 4 = 16$ — thus the name quadratic sharding.

It is hard to provide an accurate measurement for how many shards are viable today, but it is unlikely that in any foreseeable future the throughput needs of blockchain users will outgrow the limitations of quadratic sharding. The sheer number of nodes necessary to operate such a volume of shards securely is likely orders of magnitude higher than the number of nodes operating all the blockchains combined today.

1.3 State sharding

Up until now we haven't defined very well what exactly is and is not separated when a network is divided into shards. Specifically, nodes in the blockchain perform three important tasks: not only do they 1) process transactions, they also 2) relay validated transactions and completed blocks to other nodes and 3) store the state and the history of the entire network ledger. Each of these three tasks imposes a growing requirement on the nodes operating the network:

1. The necessity to process transactions requires more compute power with the increased number of transactions being processed;

2. The necessity to relay transactions and blocks requires more network bandwidth with the increased number of transactions being relayed;

3. The necessity to store data requires more storage as the state grows. Importantly, unlike the processing power and network, the storage requirement grows even if the transaction rate (number of transactions processed per second) remains constant.

From the above list it might appear that the storage requirement would be the most pressing, since it is the only one that is being increased over time even if the number of transactions per second doesn't change, but in practice the most pressing requirement today is the compute power. The entire state of Ethereum as of this writing is 100GB, easily manageable by most of the nodes. But the number of transactions Ethereum can process is around 20, orders of magnitude less than what is needed for many practical use cases.

Zilliqa is the most well-known project that shards processing but not storage. Sharding of processing is an easier problem because each node has the entire state, meaning that contracts can freely invoke other contracts and read any data from the blockchain. Some careful engineering is needed to make sure updates from multiple shards updating the same parts of the state do not conflict. In those regards Zilliqa is taking a relatively simplistic approach[2].

While sharding of storage without sharding of processing was proposed, it is extremely uncommon. Thus in practice sharding of storage, or State Sharding, almost always implies sharding of processing and sharding of network.

Practically, under State Sharding the nodes in each shard are building their own blockchain that contains transactions that affect only the local part of the global state that is assigned to that shard. Therefore, the validators in the shard only need to store their local part of the global state and only execute, and as such only relay, transactions that affect their part of the state. This partition linearly reduces the requirement on all compute power, storage, and network bandwidth, but introduces new problems, such as data availability and cross-shard transactions, both of which we will cover below.

1.4 Cross-shard transactions

The sharding model we described so far is not a very useful, because if individual shards cannot communicate with each other, they are no better than multiple independent blockchains. Even today, when sharding is not available, there's a huge demand for interoperability between various blockchains.

Let's for now only consider simple payment transactions, where each participant has account on exactly one shard. If one wishes to transfer money from

[2]Our analysis of their approach can be found here: https://medium.com/nearprotocol/8f9efae0ce3b

one account to another within the same shard, the transaction can be processed entirely by the validators in that shard. If, however, Alice that resides on shard #1 wants to send money to Bob who resides on shard #2, neither validators on shard #1(they won't be able to credit Bob's account) nor the validators on shard #2 (they won't be able to debit Alice's account) can process the entire transaction.

There are two families of approaches to cross-shard transactions:

- **Synchronous**: whenever a cross-shard transaction needs to be executed, the blocks in multiple shards that contain state transition related to the transaction get all produced at the same time, and the validators of multiple shards collaborate on executing such transactions.[3]

- **Asynchronous**: a cross-shard transaction that affects multiple shards is executed in those shards asynchronously, the "Credit" shard executing its half once it has sufficient evidence that the "Debit" shard has executed its portion. This approach tends to be more prevalent due to its simplicity and ease of coordination. This system is today proposed in Cosmos, Ethereum Serenity, Near, Kadena, and others. A problem with this approach lies in that if blocks are produced independently, there's a non-zero chance that one of the multiple blocks will be orphaned, thus making the transaction only partially applied. Consider figure 2 that depicts two shards both of which encountered a fork, and a cross-shard transaction that was recorded in blocks A and X' correspondingly. If the chains A-B and V'-X'-Y'-Z' end up being canonical in the corresponding shards, the transaction is fully finalized. If A'-B'-C'-D' and V-X become canonical, then the transaction is fully abandoned, which is acceptable. But if, for example, A-B and V-X become canonical, then one part of the transaction is finalized and one is abandoned, creating an atomicity failure. We will cover how this problem is addressed in proposed protocols in the second part, when covering changes to the fork-choice rules and consensus algorithms proposed for sharded protocols.

Note that communication between chains is useful outside of sharded blockchains too. Interoperability between chains is a complex problem that many projects are trying to solve. In sharded blockchains the problem is somewhat easier since the block structure and consensus are the same across shards, and there's a beacon chain that can be used for coordination. In a sharded blockchain, however, all the shard chains are the same, while in the global blockchains ecosystem there are lots of different blockchains, with different target use cases, decentralization and privacy guarantees.

Building a system in which a set of chains have different properties but use sufficiently similar consensus and block structure and have a common beacon chain could enable an ecosystem of heterogeneous blockchains that have a

[3]The most detailed proposal known to the authors of this document is Merge Blocks, described here: https://ethresear.ch/t/merge-blocks-and-synchronous-cross-shard-state-execution/1240

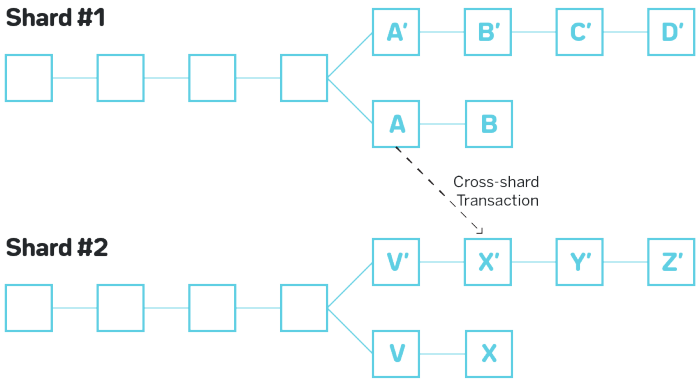

Figure 2: Asynchronous cross-shard transactions

working interoperability subsystem. Such system is unlikely to feature validator rotation, so some extra measures need to be taken to ensure security. Both Cosmos and PolkaDot are effectively such systems[4]

1.5 Malicious behavior

In this section we will review what adversarial behavior can malicious validators exercise if they manage to corrupt a shard. We will review classic approaches to avoiding corrupting shards in section 2.1.

1.5.1 Malicious forks

A set of malicious validators might attempt to create a fork. Note that it doesn't matter if the underlying consensus is BFT or not, corrupting sufficient number of validators will always make it possible to create a fork.

It is significantly more likely for more that 50% of a single shard to be corrupted, than for more than 50% of the entire network to be corrupted (we will dive deeper into these probabilities in section 2.1). As discussed in section 1.4, cross-shard transactions involve certain state changes in multiple shards, and the corresponding blocks in such shards that apply such state changes must either be all finalized (i.e. appear in the selected chains on their corresponding shards), or all be orphaned (i.e. not appear in the selected chains on their corresponding shards). Since generally the probability of shards being corrupted

[4]Refer to this writeup by Zaki Manian from Cosmos: https://forum.cosmos.network/t/polkadot-vs-cosmos/1397/2 and this tweet-storm by the first author of this document: https://twitter.com/AlexSkidanov/status/1129511266660126720 for a detailed comparison of the two

is not negligible, we can't assume that the forks won't happen even if a byzantine consensus was reached among the shard validators, or many blocks were produced on top of the block with the state change.

This problem has multiple solutions, the most common one being occasional cross-linking of the latest shard chain block to the beacon chain. The fork choice rule in the shard chains is then changed to always prefer the chain that is cross-linked, and only apply shard-specific fork-choice rule for blocks that were published since the last cross-link.

1.5.2 Approving invalid blocks

A set of validators might attempt to create a block that applies the state transition function incorrectly. For example, starting with a state in which Alice has 10 tokens and Bob has 0 tokens, the block might contain a transaction that sends 10 tokens from Alice to Bob, but ends up with a state in which Alice has 0 tokens and Bob has 1000 tokens, as shown on figure 3.

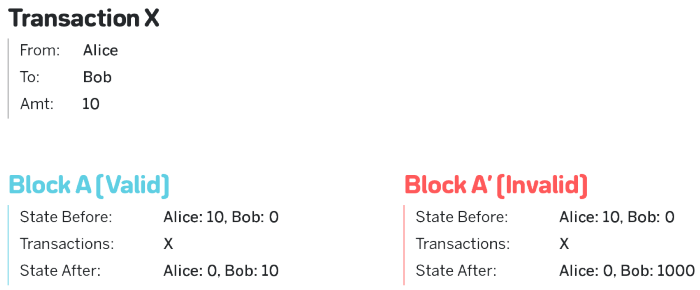

Figure 3: An example of an invalid block

In a classic non-sharded blockchain such an attack is not possible, since all the participant in the network validate all the blocks, and the block with such an invalid state transition will be rejected by both other block producers, and the participants of the network that do not create blocks. Even if the malicious validators continue creating blocks on top of such an invalid block faster than honest validators build the correct chain, thus having the chain with the invalid block being longer, it doesn't matter, since every participant that is using the blockchain for any purpose validates all the blocks, and discards all the blocks built on top of the invalid block.

On the figure 4 there are five validators, three of whom are malicious. They created an invalid block A', and then continued building new blocks on top of it. Two honest validators discarded A' as invalid and were building on top

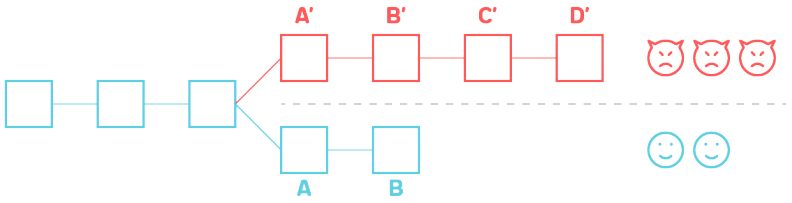

Figure 4: Attempt to create an invalid block in a non-sharded blockchain

of the last valid block known to them, creating a fork. Since there are fewer validators in the honest fork, their chain is shorter. However, in classic non-sharded blockchain every participant that uses blockchain for any purpose is responsible for validating all the blocks they receive and recomputing the state. Thus any person who has any interest in the blockchain would observe that A' is invalid, and thus also immediately discard B', C' and D', as such taking the chain A-B as the current longest valid chain.

In a sharded blockchain, however, no participant can validate all the transactions on all the shards, so they need to have some way to confirm that at no point in history of any shard of the blockchain no invalid block was included.

Note that unlike with forks, cross-linking to the Beacon chain is not a sufficient solution, since the Beacon chain doesn't have the capacity to validate the blocks. It can only validate that a sufficient number of validators in that shard signed the block (and as such attested to its correctness).

We will discuss solutions to this problem in section 2.2 below.

2 State Validity and Data Availability[5]

The core idea in sharded blockchains is that most participants operating or using the network cannot validate blocks in all the shards. As such, whenever any participant needs to interact with a particular shard they generally cannot download and validate the entire history of the shard.

The partitioning aspect of sharding, however, raises a significant potential problem: without downloading and validating the entire history of a particular shard the participant cannot necessarily be certain that the state with which

[5]This section, except for subsection 2.5.3, was previously published at https://near.ai/shard2. If you read it before, skip to the next section.

they interact is the result of some valid sequence of blocks and that such sequence of blocks is indeed the canonical chain in the shard. A problem that doesn't exist in a non-sharded blockchain.

We will first present a simple solution to this problem that has been proposed by many protocols and then analyze how this solution can break and what attempts have been made to address it.

2.1 Validators rotation

The naive solution to state validity is shown on figure 5: let's say we assume that the entire system has on the order of thousands validators, out of which no more than 20% are malicious or will otherwise fail (such as by failing to be online to produce a block). Then if we sample 200 validators, the probability of more than $\frac{1}{3}$ failing for practical purposes can be assumed to be zero.

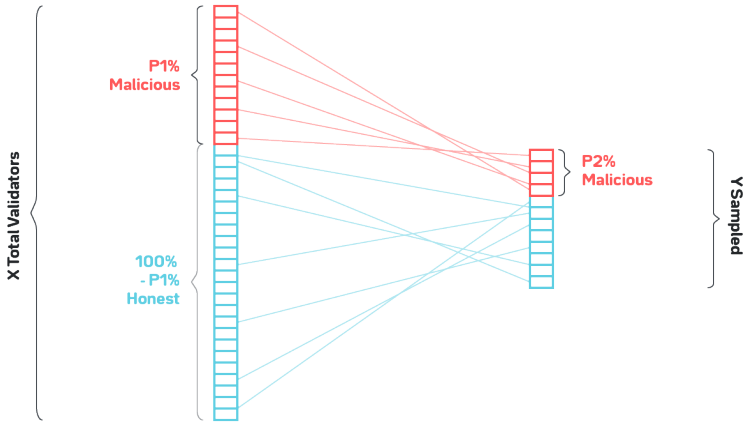

Figure 5: Sampling validators

$\frac{1}{3}$ is an important threshold. There's a family of consensus protocols, called BFT consensus protocols, that guarantees that for as long as fewer than $\frac{1}{3}$ of participants fail, either by crashing or by acting in some way that violates the protocol, the consensus will be reached.

With this assumption of honest validator percentage, if the current set of validators in a shard provides us with some block, the naive solution assumes that the block is valid and that it is built on what the validators believed to be the canonical chain for that shard when they started validating. The validators learned the canonical chain from the previous set of validators, who by the same assumption built on top of the block which was the head of the canonical chain before that. By induction the entire chain is valid, and since no set of validators at any point produced forks, the naive solution is also certain that the current chain is the only chain in the shard. See figure 6 for a visualization.

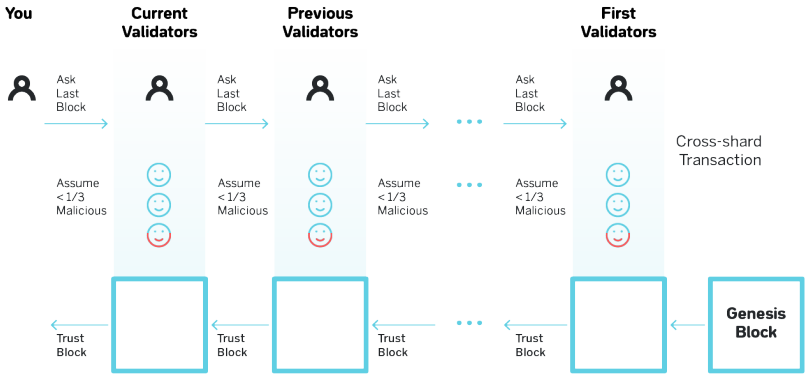

Figure 6: A blockchain with each block finalized via BFT consensus

This simple solution doesn't work if we assume that the validators can be corrupted adaptively, which is not an unreasonable assumption[6]. Adaptively corrupting a single shard in a system with 1000 shards is significantly cheaper than corrupting the entire system. Therefore, the security of the protocol decreases linearly with the number of shards. To have certainty in the validity of a block, we must know that at any point in history no shard in the system has a majority of validators colluding; with adaptive adversaries, we no longer have such certainty. As we discussed in section 1.5, colluding validators can exercise two basic malicious behaviors: create forks, and produce invalid blocks.

Malicious forks can be addressed by blocks being cross-linked to the Beacon chain that is generally designed to have significantly higher security than the shard chains. Producing invalid blocks, however, is a significantly more challenging problem to tackle.

2.2 State Validity

Consider figure 7 on which Shard #1 is corrupted and a malicious actor produces invalid block B. Suppose in this block B 1000 tokens were minted out of thin air on Alice's account. The malicious actor then produces valid block C (in a sense that the transactions in C are applied correctly) on top of B, obfuscating the invalid block B, and initiates a cross-shard transaction to Shard #2 that transfers those 1000 tokens to Bob's account. From this moment the improperly created tokens reside on an otherwise completely valid blockchain in Shard #2.

Some simple approaches to tackle this problem are:

[6]Read this article for details on how adaptive corruption can be carried out: https://medium.com/nearprotocol/d859adb464c8. For more details on adaptive corruption, read https://github.com/ethereum/wiki/wiki/Sharding-FAQ#what-are-the-security-models-that-we-are-operating-under

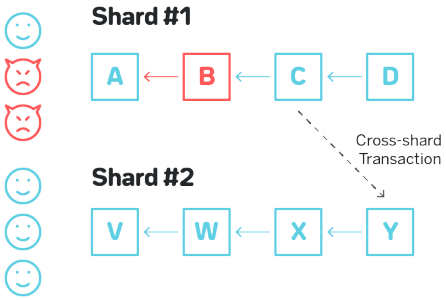

Figure 7: A cross-shard transaction from a chain that has an invalid block

1. For validators of Shard #2 to validate the block from which the transaction is initiated. This won't work even in the example above, since block C appears to be completely valid.

2. For validators in Shard #2 to validate some large number of blocks preceding the block from which the transaction is initiated. Naturally, for any number of blocks N validated by the receiving shard the malicious validators can create N+1 valid blocks on top of the invalid block they produced.

A promising idea to resolve this issue would be to arrange shards into an undirected graph in which each shard is connected to several other shards, and only allow cross-shard transactions between neighboring shards (e.g. this is how Vlad Zamfir's sharding essentially works[7], and similar idea is used in Kadena's Chainweb [1]). If a cross-shard transaction is needed between shards that are not neighbors, such transaction is routed through multiple shards. In this design a validator in each shard is expected to validate both all the blocks in their shard as well as all the blocks in all the neighboring shards. Consider a figure below with 10 shards, each having four neighbors, and no two shards requiring more than two hops for a cross-shard communication shown on figure 8.

Shard #2 is not only validating its own blockchain, but also blockchains of all the neighbors, including Shard #1. So if a malicious actor on Shard #1 is attempting to create an invalid block B, then build block C on top of it and initiate a cross-shard transaction, such cross-shard transaction will not go through since Shard #2 will have validated the entire history of Shard #1 which will cause it to identify invalid block B.

[7]Read more about the design here: https://medium.com/nearprotocol/37e538177ed9

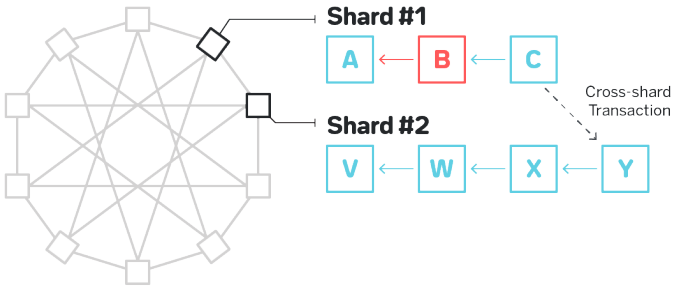

Figure 8: An invalid cross-shard transaction in chainweb-like system that will get detected

While corrupting a single shard is no longer a viable attack, corrupting a few shards remains a problem. On figure 9 an adversary corrupting both Shard #1 and Shard #2 successfully executes a cross-shard transaction to Shard #3 with funds from an invalid block B:

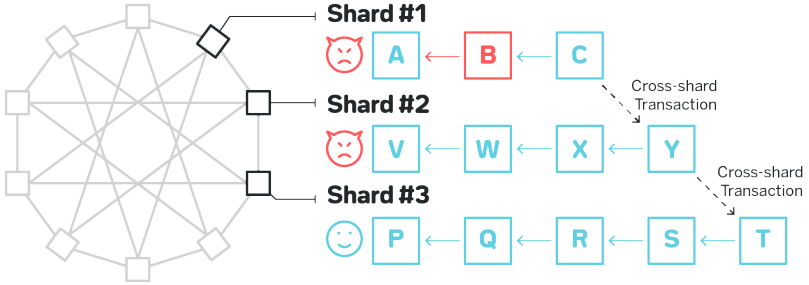

Figure 9: An invalid cross-shard transaction in chainweb-like system that will not get detected

Shard #3 validates all the blocks in Shard #2, but not in Shard #1, and has no way to detect the malicious block.

There are two major directions of properly solving state validity: fishermen

and cryptographic proofs of computation.

2.3 Fisherman

The idea behind the first approach is the following: whenever a block header is communicated between chains for any purpose (such as cross-linking to the beacon chain, or a cross-shard transaction), there's a period of time during which any honest validator can provide a proof that the block is invalid. There are various constructions that enable very succinct proofs that the blocks are invalid, so the communication overhead for the receiving nodes is way smaller than that of receiving a full block.

With this approach for as long as there's at least one honest validator in the shard, the system is secure.

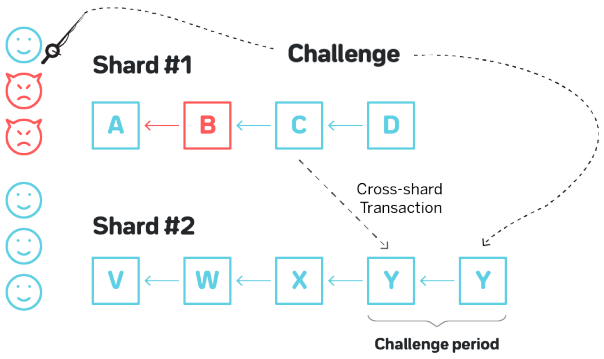

Figure 10: Fisherman

This is the dominant approach (besides pretending the problem doesn't exist) among the proposed protocols today. This approach, however, has two major disadvantages:

1. The challenge period needs to be sufficiently long for the honest validator to recognize a block was produced, download it, fully verify it, and prepare the challenge if the block is invalid. Introducing such a period would significantly slow down the cross-shard transactions.

2. The existence of the challenge protocol creates a new vector of attacks when malicious nodes spam with invalid challenges. An obvious solution to this problem is to make challengers deposit some amount of tokens that are returned if the challenge is valid. This is only a partial solution, as it might still be beneficial for the adversary to spam the system (and burn the deposits) with invalid challenges, for example to prevent the valid

challenge from a honest validator from going through. These attacks are called **Grieving Attacks**.

See section 3.7.2 for a way to get around the latter point.

2.4 Succinct Non-interactive Arguments of Knowledge

The second solution to multiple-shard corruption is to use some sort of cryptographic constructions that allow one to prove that a certain computation (such as computing a block from a set of transactions) was carried out correctly. Such constructions do exist, e.g. zk-SNARKs, zk-STARKs and a few others, and some are actively used in blockchain protocols today for private payments, most notably ZCash. The primary problem with such primitives is that they are notoriously slow to compute. E.g. Coda Protocol, that uses zk-SNARKs specifically to prove that all the blocks in the blockchain are valid, said in one of the interviews that it can take 30 seconds per transaction to create a proof (this number is probably smaller by now).

Interestingly, a proof doesn't need to be computed by a trusted party, since the proof not only attests to the validity of the computation it is built for, but to the validity of the proof itself. Thus, the computation of such proofs can be split among a set of participants with significantly less redundancy than would be necessary to perform some trustless computation. It also allows for participants who compute zk-SNARKs to run on special hardware without reducing the decentralization of the system.

The challenges of zk-SNARKs, besides performance, are:

1. Dependency on less-researched and less-time-tested cryptographic primitives;
2. "Toxic waste" — zk-SNARKs depend on a trusted setup in which a group of people performs some computation and then discards the intermediate values of that computation. If all the participants of the procedure collude and keep the intermediate values, fake proofs can be created;
3. Extra complexity introduced into the system design;
4. zk-SNARKs only work for a subset of possible computations, so a protocol with a Turing-complete smart contract language wouldn't be able to use SNARKs to prove the validity of the chain.

2.5 Data Availability

The second problem we will touch upon is data availability. Generally nodes operating a particular blockchain are separated into two groups: Full Nodes, those that download every full block and validate every transaction, and **Light Nodes**, those that only download block headers, and use Merkle proofs for parts of the state and transactions they are interested in, as shown on figure 11.

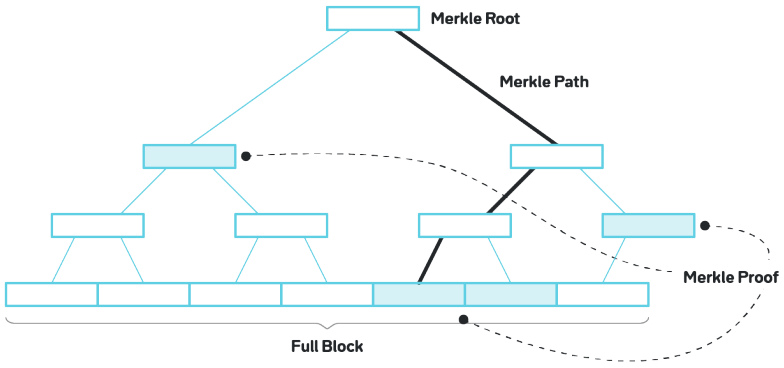

Figure 11: Merkle Tree

Now if a majority of full nodes collude, they can produce a block, valid or invalid, and send its hash to the light nodes, but never disclose the full content of the block. There are various ways they can benefit from it. For example, consider figure 12:

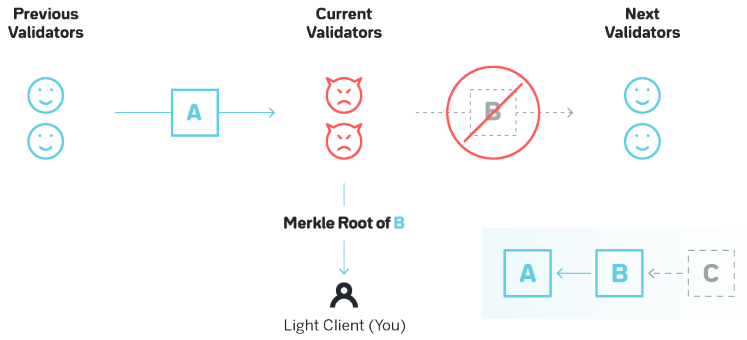

Figure 12: Data Availability problem

There are three blocks: the previous, A, is produced by honest validators; the current, B, has validators colluding; and the next, C, will be also produced by honest validators (the blockchain is depicted in the bottom right corner).

You are a merchant. The validators of the current block (B) received block A from the previous validators, computed a block in which you receive money,

and sent you a header of that block with a Merkle proof of the state in which you have money (or a Merkle proof of a valid transaction that sends the money to you). Confident the transaction is finalized, you provide the service.

However, the validators never distribute the full content of the block B to anyone. As such, the honest validators of block C can't retrieve the block, and are either forced to stall the system or to build on top of A, depriving you as a merchant of money.

When we apply the same scenario to sharding, the definitions of full and light node generally apply per shard: validators in each shard download every block in that shard and validate every transaction in that shard, but other nodes in the system, including those that snapshot shard chains state into the beacon chain, only download the headers. Thus the validators in the shard are effectively **full nodes** for that shard, while other participants in the system, including the beacon chain, operate as **light nodes**.

For the fisherman approach we discussed above to work, honest validators need to be able to download blocks that are cross-linked to the beacon chain. If malicious validators cross-linked a header of an invalid block (or used it to initiate a cross-shard transaction), but never distributed the block, the honest validators have no way to craft a challenge.

We will cover three approaches to address this problem that complement each other.

2.5.1 Proofs of Custody

The most immediate problem to be solved is whether a block is available once it is published. One proposed idea is to have so-called Notaries that rotate between shards more often than validators whose only job is to download a block and attest to the fact that they were able to download it. They can be rotated more frequently because they don't need to download the entire state of the shard, unlike the validators who cannot be rotated frequently since they must download the state of the shard each time they rotate, as shown on figure 13.

The problem with this naive approach is that it is impossible to prove later whether the Notary was or was not able to download the block, so a Notary can choose to always attest that they were able to download the block without even attempting to retrieve it. One solution to this is for Notaries to provide some evidence or to stake some amount of tokens attesting that the block was downloaded. One such solution is discussed here: https://ethresear.ch/t/1-bit-aggregation-friendly-custody-bonds/2236.

2.5.2 Erasure Codes

When a particular light node receives a hash of a block, to increase the node's confidence that the block is available it can attempt to download a few random pieces of the block. This is not a complete solution, since unless the light nodes collectively download the entire block the malicious block producers can choose

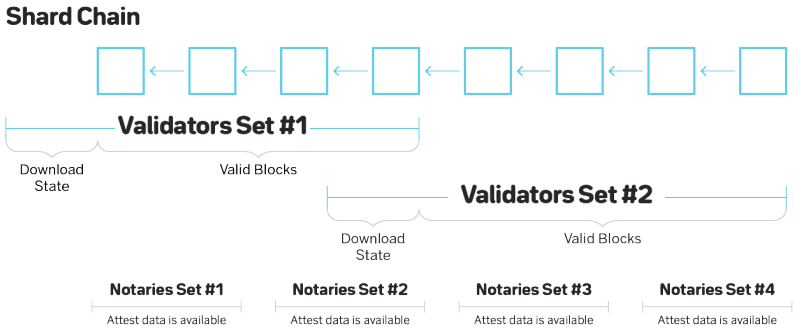

Figure 13: Validators need to download state and thus cannot be rotated frequently

to withhold the parts of the block that were not downloaded by any light node, thus still making the block unavailable.

One solution is to use a construction called Erasure Codes to make it possible to recover the full block even if only some part of the block is available, as shown on figure 14.

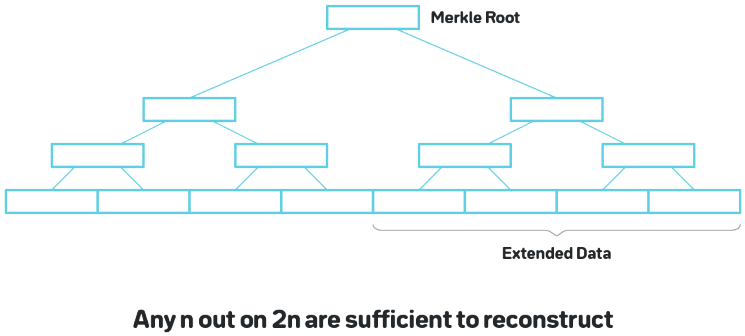

Figure 14: Merkle tree built on top of erasure coded data

Both Polkadot and Ethereum Serenity have designs around this idea that provide a way for light nodes to be reasonably confident the blocks are available. The Ethereum Serenity approach has a detailed description in [2].

2.5.3 Polkadot's approach to data availability

In Polkadot, like in most sharded solutions, each shard (called parachain) snapshots its blocks to the beacon chain (called relay chain). Say there are $2f + 1$ validators on the relay chain. The block producers of the parachain blocks, called collators, once the parachain block is produced compute an erasure coded version of the block that consists of $2f + 1$ parts such that any f parts are sufficient to reconstruct the block. They then distribute one part to each validator on the relay chain. A particular relay chain validator would only sign on a relay chain block if they have their part for each parachain block that is snapshotted to such relay chain block. Thus, if a relay chain block has signatures from $2f + 1$ validators, and for as long as no more than f of them violated the protocol, each parachain block can be reconstructed by fetching the parts from the validators that follow the protocol. See figure 15.

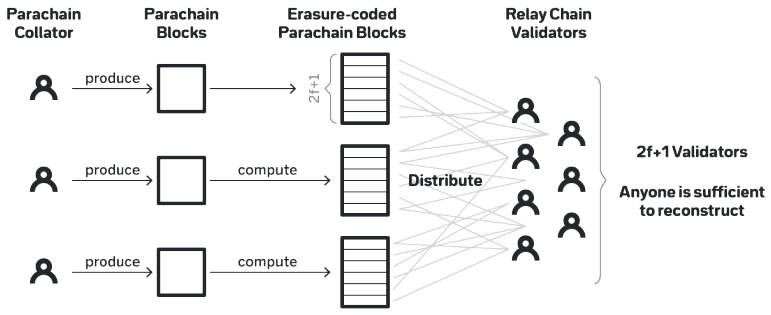

Figure 15: Polkadot's data availability

2.5.4 Long term data availability

Note that all the approaches discussed above only attest to the fact that a block was published at all, and is available now. Blocks can later become unavailable for a variety of reasons: nodes going offline, nodes intentionally erasing historical data, and others.

A whitepaper worth mentioning that addresses this issue is Polyshard [3], which uses erasure codes to make blocks available across shards even if several shards completely lose their data. Unfortunately their specific approach requires all the shards to download blocks from all other shards, which is prohibitively expensive.

The long term availability is not as pressing of an issue: since no participant in the system is expected to be capable of validating all the chains in all the

shards, the security of the sharded protocol needs to be designed in such a way that the system is secure even if some old blocks in some shards become completely unavailable.

3 Nightshade

3.1 From shard chains to shard chunks

The sharding model with shard chains and a beacon chain is very powerful but has certain complexities. In particular, the fork choice rule needs to be executed in each chain separately, the fork choice rule in the shard chains and the beacon chain must be built differently and tested separately.

In Nightshade we model the system as a single blockchain, in which each block logically contains all the transactions for all the shards, and changes the whole state of all the shards. Physically, however, no participant downloads the full state or the full logical block. Instead, each participant of the network only maintains the state that corresponds to the shards that they validate transactions for, and the list of all the transactions in the block is split into physical chunks, one chunks per shard.

Under ideal conditions each block contains exactly one chunk per shard per block, which roughly corresponds to the model with shard chains in which the shard chains produce blocks with the same speed as the beacon chain. However, due to network delays some chunks might be missing, so in practice each block contains either one or zero chunks per shard. See section 3.3 for details on how blocks are produced.

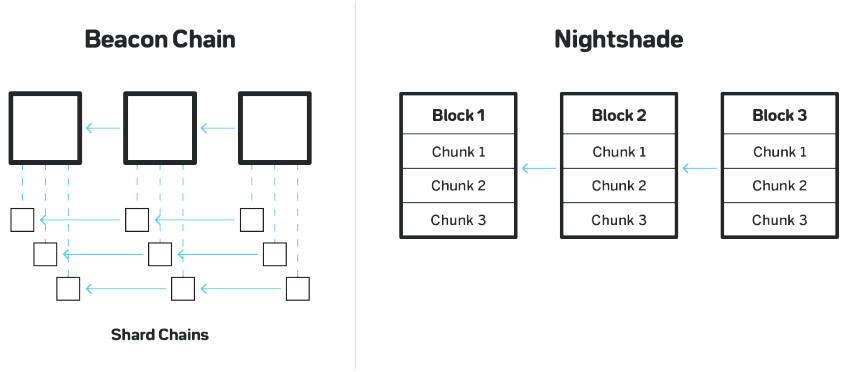

Figure 16: A model with shard chains on the left and with one chain having blocks split into chunks on the right

3.2 Consensus

The two dominant approaches to the consensus in the blockchains today are the longest (or heaviest) chain, in which the chain that has the most work or stake used to build it is considered canonical, and BFT, in which for each block some set of validators reach a BFT consensus.

In the protocols proposed recently the latter is a more dominant approach, since it provides immediate finality, while in the longest chain more blocks need to be built on top of the block to ensure the finality. Often for a meaningful security the time it takes for sufficient number of blocks to be built takes on the order of hours.

Using BFT consensus on each block also has disadvantages, such as:

1. BFT consensus involves considerable amount of communication. While recent advances allow the consensus to be reached in linear time in number of participants (see e.g. [4]), it is still noticeable overhead per block;

2. It is unfeasible for all the network participants to participate in the BFT consensus per block, thus usually only a randomly sampled subset of participants reach the consensus. A randomly sampled set can be, in principle, adaptively corrupted, and a fork in theory can be created. The system either needs to be modelled to be ready for such an event, and thus still have a fork-choice rule besides the BFT consensus, or be designed to shut down in such an event. It is worth mentioning that some designs, such as Algorand [5], significantly reduce the probability of adaptive corruption.

3. Most importantly, the system stalls if $\frac{1}{3}$ or more of all the participants are offline. Thus, any temporary network glitch or a network split can completely stall the system. Ideally the system must be able to continue to operate for as long as at least half of the participants are online (heaviest chain-based protocols continue operating even if less than half of the participants are online, but the desirability of this property is more debatable within the community).

A hybrid model in which the consensus used is some sort of the heaviest chain, but some blocks are periodically finalized using a BFT finality gadget maintain the advantages of both models. Such BFT finality gadgets are Casper FFG [6] used in Ethereum 2.0 [8], Casper CBC (see https://vitalik.ca/general/2018/12/05/cbc_casper.html) and GRANDPA (see https://medium.com/polkadot-network/d08a24a021b5) used in Polkadot.

Nightshade uses the heaviest chain consensus. Specifically when a block producer produces a block (see section 3.3), they can collect signatures from other block producers and validators attesting to the previous block. See section 3.8 for details how such large number of signatures is aggregated. The weight

[8] Also see the whiteboard session with Justin Drake for an indepth overview of Casper FFG, and how it is integrated with the GHOST heaviest chain consensus here: https://www.youtube.com/watch?v=S262StTwkmo

of a block is then the cumulative stake of all the signers whose signatures are included in the block. The weight of a chain is the sum of the block weights.

On top of the heaviest chain consensus we use a finality gadget that uses the attestations to finalize the blocks. To reduce the complexity of the system, we use a finality gadget that doesn't influence the fork choice rule in any way, and instead only introduces extra slashing conditions, such that once a block is finalized by the finality gadget, a fork is impossible unless a very large percentage of the total stake is slashed. Casper CBC is such a finality gadget, and we presently model with Casper CBC in mind.

We also work on a separate BFT protocol called TxFlow. At the time of writing this document it is unclear if TxFlow will be used instead of Casper CBC. We note, however, that the choice of the finality gadget is largely orthogonal to the rest of the design.

3.3 Block production

In Nightshade there are two roles: block producers and validators. At any point the system contains w block producers, $w = 100$ in our models, and wv validators, in our model $v = 100$, $wv = 10,000$. The system is Proof-of-Stake, meaning that both block producers and validators have some number of internal currency (referred to as "tokens") locked for a duration of time far exceeding the time they spend performing their duties of building and validating the chain.

As with all the Proof of Stake systems, not all the w block producers and not all the wv validators are different entities, since that cannot be enforced. Each of the w block producers and the wv validators, however, do have a separate stake.

The system contains n shards, $n = 1000$ in our model. As mentioned in section 3.1, in Nightshade there are no shard chains, instead all the block producers and validators are building a single blockchain, that we refer to as *the main chain*. The state of the main chain is split into n shards, and each block producer and validator at any moment only have downloaded locally a subset of the state that corresponds to some subset of the shards, and only process and validate transactions that affect those parts of the state.

To become a block producer, a participant of the network locks some large amount of tokens (a stake). The maintenance of the network is done in epochs, where an epoch is a period of time on the order of days. The participants with the w largest stakes at the beginning of a particular epoch are the block producers for that epoch. Each block producer is assigned to s_w shards, (say $s_w = 40$, which would make $s_w w/n = 4$ block producers per shard). The block producer downloads the state of the shard they are assigned to before the epoch starts, and throughout the epoch collects transactions that affect that shard, and applies them to the state.

For each block b on the main chain, and for every shards s, there's one of the assigned block producers to s who is responsible to produce the part of b related to the shard. The part of b related to shard s is called a *chunk*, and contains the list of the transactions for the shard to be included in b, as well as the merkle

root of the resulting state. b will ultimately only contain a very small header of the chunk, namely the merkle root of all the applied transactions (see section 3.7.1 for exact details), and the merkle root of the final state.

Throughout the rest of the document we often refer to the block producer that is responsible to produce a chunk at a particular time for a particular shard as a *chunk producer*. Chunk producer is always one of the block producers.

The block producers and the chunk producers rotate each block according to a fixed schedule. The block producers have an order and repeatedly produce blocks in that order. E.g. if there are 100 block producers, the first block producers is responsible for producing blocks 1, 101, 201 etc, the second is responsible for producing 2, 102, 202 etc).

Since chunk production, unlike the block production, requires maintaining the state, and for each shard only $s_w w/n$ block producers maintain the state per shard, correspondingly only those $s_w w/n$ block producers rotate to create chunks. E.g. with the constants above with four block producers assigned to each shard, each block producer will be creating chunks once every four blocks.

3.4 Ensuring data availability

To ensure the data availability we use an approach similar to that of Polkadot described in section 2.5.3. Once a block producer produces a chunk, they create an erasure coded version of it with an optimal $(w, \lfloor w/6 + 1 \rfloor)$ block code of the chunk.

They then send one piece of the erasure coded chunk (we call such pieces *chunk parts*, or just *parts*) to each block producer.

We compute a merkle tree that contains all the parts as the leaves, and the header of each chunk contains the merkle root of such tree.

The parts are sent to the validators via *onepart* messages. Each such message contains the chunk header, the ordinal of the part and the part contents. The message also contains the signature of the block producer who produced the chunk and the merkle path to prove that the part corresponds to the header and is produced by the proper block producer.

Once a block producer receives a main chain block, they first check if they have *onepart* messages for each chunk included in the block. If not, the block is not processed until the missing *onepart* messages are retrieved.

Once all the *onepart* messages are received, the block producer fetches the remaining parts from the peers and reconstructs the chunks for which they hold the state.

The block producer doesn't process a main chain block if for at least one chunk included in the block they don't have the corresponding *onepart* message, or if for at least one shard for which they maintain the state they cannot reconstruct the entire chunk.

For a particular chunk to be available it is enough that $\lfloor w/6 \rfloor + 1$ of the block producers have their parts and serve them. Thus, for as long as the number of malicious actors doesn't exceed $\lfloor w/3 \rfloor$ no chain that has more than half block producers building it can have unavailable chunks.

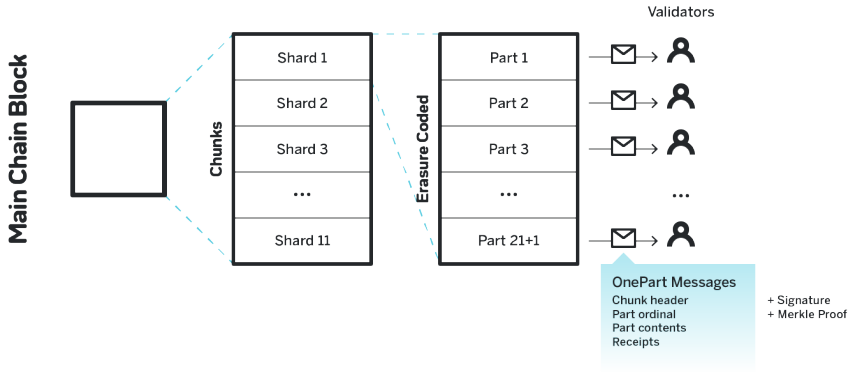

Figure 17: Each block contains one or zero chunks per shard, and each chunk is erasure coded. Each part of the erasure coded chunk is sent to a designated block producer via a special *onepart* message

3.4.1 Dealing with lazy block producers

If a block producer has a block for which a *onepart* message is missing, they might choose to still sign on it, because if the block ends up being on chain it will maximize the reward for the block producer. There's no risk for the block producer since it is impossible to prove later that the block producer didn't have the *onepart* message.

To address it we make each chunk producer when creating the chunk to choose a color (red or blue) for each part of the future encoded chunk, and store the bitmask of assigned color in the chunk before it is encoded. Each *onepart* message then contains the color assigned to the part, and the color is used when computing the merkle root of the encoded parts. If the chunk producer deviates from the protocol, it can be easily proven, since either the merkle root will not correspond to *onepart* messages, or the colors in the *onepart* messages that correspond to the merkle root will not match the mask in the chunk.

When a block producer signs on a block, they include a bitmask of all the red parts they received for the chunks included in the block. Publishing an incorrect bitmask is a slashable behavior. If a block producer hasn't received a *onepart* message, they have no way of knowing the color of the message, and thus have a 50% chance of being slashed if they attempt to blidnly sign the block.

3.5 State transition application

The chunk producers only choose which transactions to include in the chunk but do not apply the state transition when they produce a chunk. Correspondingly,

the chunk header contains the merkle root of the merkelized state as of *before* the transactions in the chunk are applied.

The transactions are only applied when a full block that includes the chunk is processed. A participant only processes a block if

1. The previous block was received and processed;
2. For each chunk the participant doesn't maintain the state for they have seen the *onepart* message;
3. For each chunk the participant does maintain the state for they have the full chunk.

Once the block is being processed, for each shard for which the participant maintains the state for, they apply the transactions and compute the new state as of after the transactions are applied, after which they are ready to produce the chunks for the next block, if they are assigned to any shard, since they have the merkle root of the new merkelized state.

3.6 Cross-shard transactions and receipts

If a transaction needs to affect more than one shard, it needs to be consecutively executed in each shard separately. The full transaction is sent to the first shard affected, and once the transaction is included in the chunk for such shard, and is applied after the chunk is included in a block, it generates a so called *receipt* transaction, that is routed to the next shard in which the transaction need to be executed. If more steps are required, the execution of the receipt transaction generates a new receipt transaction and so on.

3.6.1 Receipt transaction lifetime

It is desirable that the receipt transaction is applied in the block that immediately follows the block in which it was generated. The receipt transaction is only generated after the previous block was received and applied by block producers that maintain the originating shard, and needs to be known by the time the chunk for the next block is produced by the block producers of the destination shard. Thus, the receipt must be communicated from the source shard to the destination shard in the short time frame between those two events.

Let A be the last produced block which contains a transaction t that generates a receipt r. Let B be the next produced block (i.e. a block that has A as its previous block) that we want to contain r. Let t be in the shard a and r be in the shard b.

The lifetime of the receipt, also depicted on figure 18, is the following:

Producing and storing the receipts. The chunk producer cp_a for shard a receives the block A, applies the transaction t and generates the receipt r. cp_a then stores all such produced receipts in its internal persistent storage indexed by the source shard id.

Distributing the receipts. Once cp_a is ready to produce the chunk for shard a for block B, they fetch all the receipts generated by applying the transactions from block A for shard a, and included them into the chunk for shrad a in block B. Once such chunk is generated, cp_a produces its erasure coded version and all the corresponding *onepart* messages. cp_a knows what block producers maintain the full state for which shards. For a particular block producer bp cp_a includes the receipts that resulted from applying transactions in block A for shard a that have any of the shards that bp cares about as their destination in the *onepart* message when they distributed the chunk for shard a in block B (see figure 17, that shows receipts included in the *onepart* message).

Receiving the receipts. Remember that the participants (both block producers and validators) do not process blocks until they have *onepart* messages for each chunk included in the block. Thus, by the time any particular particpiant applies the block B, they have all the *onepart* messages that correspond to chunks in B, and thus they have all the incoming receipts that have the shards the participant maintains state for as their destination. When applying the state transition for a particular shard, the participant apply both the receipts that they have collected for the shard in the *onepart* messages, as well as all the transactions included in the chunk itself.

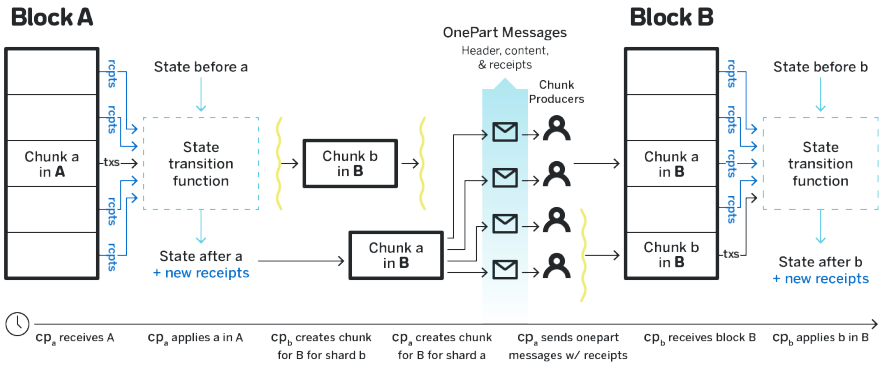

Figure 18: The lifetime of a receipt transaction

3.6.2 Handling too many receipts

It is possible that the number of receipts that target a particular shard in a particular block is too large to be processed. For example, consider figure 19, in which each transaction in each shard generates a receipt that targets shard 1. By the next block the number of receipts that shard 1 needs to process is comparable to the load that all the shards combined processed while handling the previous block.

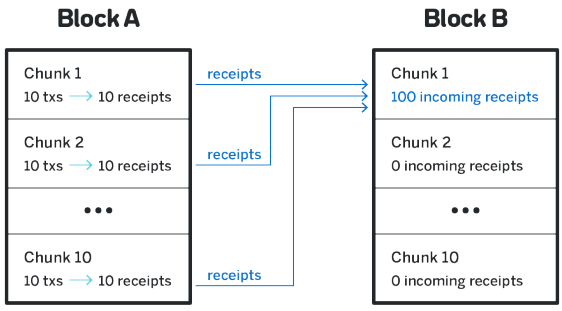

Figure 19: If all the receipts target the same shard, the shard might not have the capacity to process them

To address it we use a technique similar to that used in QuarkChain [9]. Specifically, for each shard the last block B and the last shard s within that block from which the receipts were applied is recorded. When the new shard is created, the receipt are applied in order first from the remaining shards in B, and then in blocks that follow B, until the new chunk is full. Under normal circumstances with a balanced load it will generally result in all the receipts being applied (and thus the last shard of the last block will be recorded for each chunk), but during times when the load is not balanced, and a particular shard receives disproportionately many receipts, this technique allows them to be processed while respecting the limits on the number of transactions included.

Note that if such unbalanced load remains for a long time, the delay from the receipt creation until application can continue growing indefinitely. One way to address it is to drop any transaction that creates a receipt targeting a shard that has a processing delay that exceeds some constant (e.g. one epoch).

Consider figure 20. By block B the shard 4 cannot process all the receipts, so it only processes receipts origination from up to shard 3 in block A, and records it. In block C the receipts up to shard 5 in block B are included, and then by block D the shard catches up, processing all the remaining receipts in block B and all the receipts from block C.

3.7 Chunks validation

A chunk produced for a particular shard (or a shard block produced for a particular shard chain in the model with shard chains) can only be validated by the

[9] See the whiteboard episode with QuarkChain here: https://www.youtube.com/watch?v=opEtG6NM4x4, in which the approach to cross-shard transactions is discussed, among other things

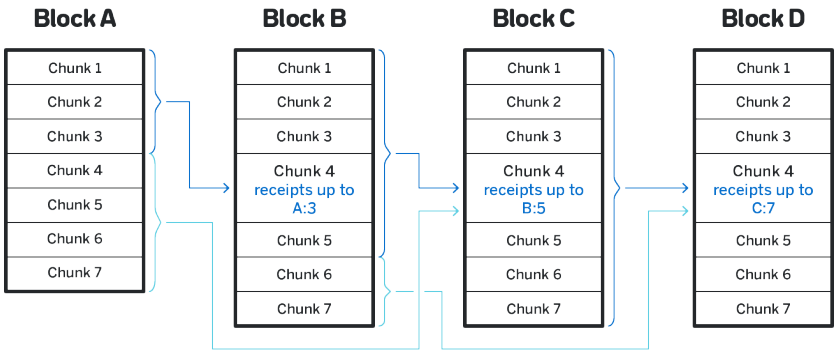

Figure 20: Delayed receipts processing

participants that maintain the state. They can be block producers, validators, or just external witnesses that downloaded the state and validate the shard in which they store assets.

In this document we assume that majority of the participants cannot store the state for a large fraction of the shards. It is worth mentioning, however, that there are sharded blockchains that are designed with the assumption that most participants do have capacity to store the state for and validate most of the shards, such as QuarkChain.

Since only a fraction of the participants have the state to validate the shard chunks, it is possible to adaptive corrupt just the participants that have the state, and apply an invalid state transition.

Multiple sharding designs were proposed that sample validators every few days, and within a day any block in the shard chain that has more than $2/3$ of signatures of the validators assigned to such shard is immediately considered final. With such approach an adaptive adversary only needs to corrupt $2n/3+1$ of the validators in a shard chain to apply an invalid state transition, which, while is likely hard to pull off, is not a level of security sufficient for a public blockchain.

As discussed in section 2.3, the common approach is to allow a certain window of time after a block is created for any participant that has state (whether it's a block producer, a validator, or an external observer) to challenge its validity. Such participants are called Fishermen. For a fisherman to be able to challenge an invalid block, it must be ensured that such a block is available to them. The data availability in Nightshade is discussed in section 3.4.

In Nightshade once a block is produced, the chunks were not validated by anyone but the actual chunk producer. In particular, the block producer that suggested the block naturally didn't have the state for most of the shards, and

was not able to validate the chunks. When the next block is produced, it contains attestations (see section 3.2) of multiple block producers and validators, but since the majority of block producers and validators do not maintain state for most shards as well, a block with just one invalid chunk will collect significantly more than half of the attestations and will continue being on the heaviest chain.

To address this issue, we allow any participant that maintains the state of a shard to submit a challenge on-chain for any invalid chunk produced in that shard.

3.7.1 State validity challenge

Once a participant detects that a particular chunk is invalid, they need to provide a proof that the chunk is invalid. Since the majority of the network participants do not maintain the state for the shard in which the invalid chunk is produced, the proof needs to have sufficient information to confirm the block is invalid without having the state.

We set a limit L_s of the amount of state (in bytes) that a single transaction can cumulatively read or write. Any transaction that touches more than L_s state is considered to be invalid. Remember from the section 3.5 that the chunk in a particular block B only contains the transactions to be applied, but not the new state root. The state root included in the chunk in block B is the state root before applying such transactions, but after applying the transactions from the last chunk in the same shard before the block B. A malicious actor that wishes to apply an invalid state transition would include an incorrect state root in block B that doesn't correspond to the state root that results from applying the transactions in the preceding chunk.

We extend the information that a chunk producer includes in the chunk. Instead of just including the state after applying all the transactions, it instead includes a state root after applying each contiguous set of transactions that collectively read and write L_s bytes of state. With this information for the fisherman to create a challenge that a state transition is applied incorrectly it is sufficient to find the first such invalid state root, and include just L_s bytes of state that are affected by the transactions between the last state root (which was valid) and the current state root with the merkle proofs. Then any participant can validate the transactions in the segment and confirm that the chunk is invalid.

Similarly, if the chunk producer attempted to include transactions that read and write more than L_s bytes of state, for the challenge it is enough to include the first L_s bytes it touches with the merkle proofs, which will be enough to apply the transactions and confirm that there's a moment when an attempt to read or write content beyond L_s bytes is made.

3.7.2 Fishermen and fast cross-shard transactions

As discussed in section 2.3, once we assume that the shard chunks (or shard blocks in the model with shard chains) can be invalid and introduce a challenge period, it negatively affects the finality, and thus cross-shard communication. In particular, the destination shard of any cross-shard transction cannot be certain the originating shard chunk or block is final until the challenge period is over (see figure 21).

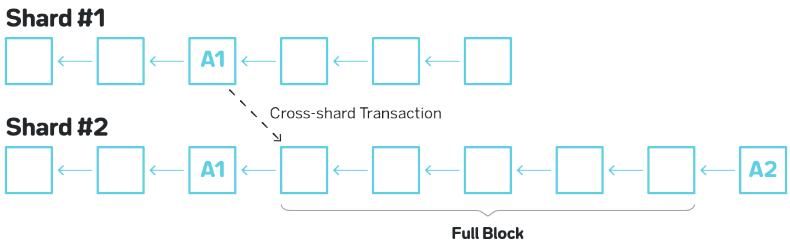

Figure 21: Waiting for the challenge period before applying a receipt

The way to address it in a way that makes the cross-shard transactions instantenious is for the destination shard to not wait for the challenge period after the source shard transaction is published, and apply the receipt transaction immediately, but then roll back the destination shard together with the source shard if later the originating chunk or block was found to be invalid (see figure 22). This applies very naturally to the Nightshade design in which the shard chains are not independent, but instead the shard chunks are all published together in the same main chain block. If any chunk is found to be invalid, the entire block with that chunk is considered invalid, and all the blocks built on top of it. See figure 23.

Both of the above approaches provide atomicity assuming that the challenge period is sufficiently long. We use the latter approach since providing fast cross-shard transactions under normal circumstances outweights the inconvenience of the destination shard rolling back due to an invalid state transition in one of the source shards, which is an extremely rare event.

3.7.3 Hiding validators

The existence of the challenges already significantly reduces the probability of adaptive corruption, since to finalize a chunk with an invalid state transition post

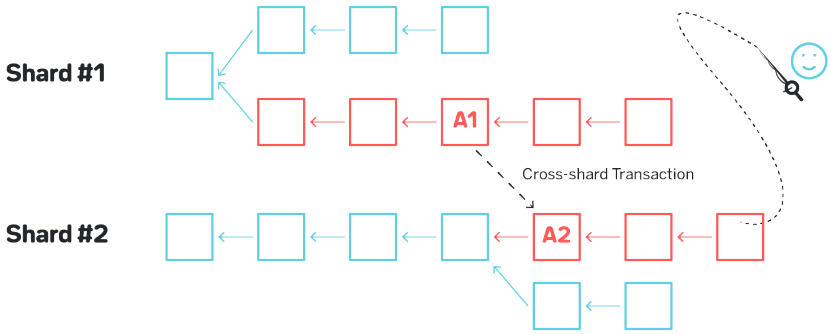

Figure 22: Applying receipts immediately and rolling back the destination chain if the source chain had an invalid block

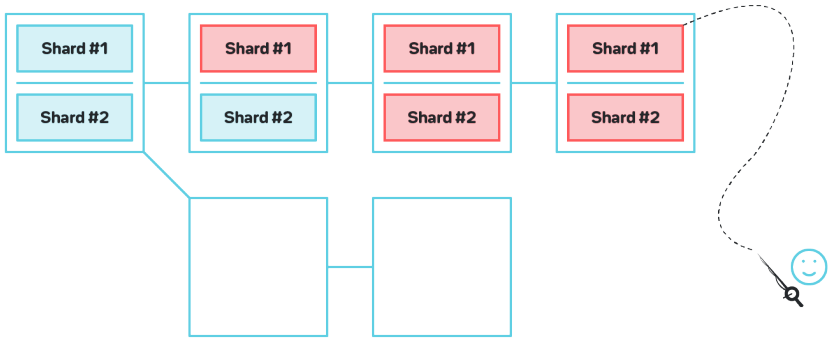

Figure 23: Fisherman challenge in Nightshade

the challenge period the adaptive adversary needs to corrupt **all** the participants that maintain the state of the shard, including all the validators.

Estimating the likelihood of such an event is extremely complex, since no sharded blockchain has been live sufficiently long for any such attack to be attempted. We argue that the probability, while extremely low, is still sufficiently large for a system that is expected to execute multi-million transactions and run a world-wide financial operations.

There are two main reasons for this belief:

1. Most of the validators of the Proof-of-Stake chains and miners of the

Proof-of-Work chains are primarily incentivized by the financial upside. If an adaptive adversary offers them more money then the expected return from operating honestly, it is reasonable to expect that many validators will accept the offer.

2. Many entities do validation of Proof-of-Stake chains professionally, and it is expected that a large percentage of the stake in any chain will be from such entities. The number of such entities is sufficiently small for an adaptive adversary to get to know most of them personally and have a good understanding of their inclanation to be corrupted.

We take one step further in reducing the probability of the adaptive corruption by hiding which validators are assigned to which shard. The idea is remotely similar to the way Algorand [5] conceals validators.

It is critical to note that even if the validators are concealed, as in Algorand or as described below, the adaptive corruption is still in theory possible. While the adaptive adversary doesn't know the participants that will create or validate a block or a chunk, the participants themselves do know that they will perform such a task and have a cryptographic proof of it. Thus, the adversary can broadcast their intent to corrupt, and pay to any participant that will provide such a cryptographic proof. We note however, that since the adversary doesn't know the validators that are assigned to the shard they want to corrupt, they have no other choice but to broadcast their intent to corrupt a particular shard to the entire community. At that point it is economically beneficial for any honest participant to spin up a full node that validates that shard, since there's a high chance of an invalid block appearing in that shard, which is an opportunity to create a challenge and collect associated reward.

To not reveal the validators that are assigned to a particular shard, we do the following (see figure 24):

Using VRF to get the assignment. At the beginning of each epoch each validator uses a VRF to get a bitmask of the shards the validator is assigned to. The bitmask of each validator will have S_w bits (see section 3.3 for the definition of S_w). The validator then fetches the state of the corresponding shards, and during the epoch for each block received validates the chunks that correspond to the shards that the validator is assigned to.

Sign on blocks instead of chunks. Since the shards assignment is concealed, the validator cannot sign on chunks. Instead it always signs on the entire block, thus not revealing what shards it validates. Specifically, when the validator receives a block and validates all the chunks, it either creates a message that attests that all the chunks in all the shards the validator is assigned to are valid (without indicating in any way what those shards are), or a message that contains a proof of an invalid state transition if any chunk is invalid. See the section 3.8 for the details on how such messages are aggregated, section 3.7.4 for the details on how to prevent validators from piggy-backing on messages from other validators, and section 3.7.5 for the details how to reward and punish validators should a successful invalid state transition challenge actually happen.

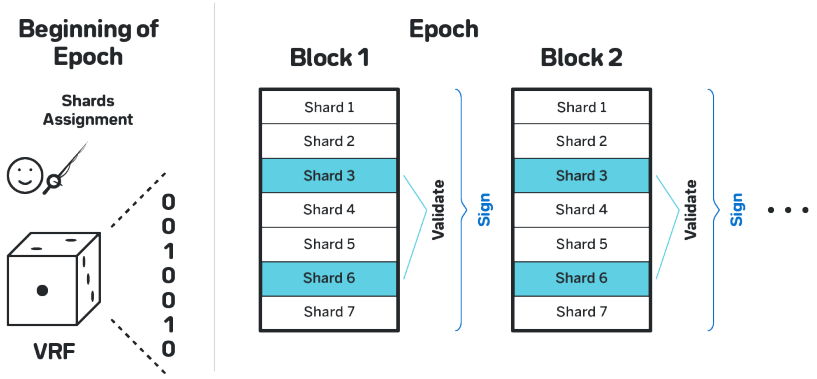

Figure 24: Concealing the validators in Nightshade

3.7.4 Commit-Reveal

One of the common problems with validators is that a validator can skip downloading the state and actually validating the chunks and blocks, and instead observe the network, see what the other validators submit and repeat their messages. A validator that follows such a strategy doesn't provide any extra security for the network, but collects rewards.

A common solution for this problem is for each validator to provide a proof that they actually validated the block, for example by providing a unique trace of applying the state transition, but such proofs significantly increase the cost of validation.

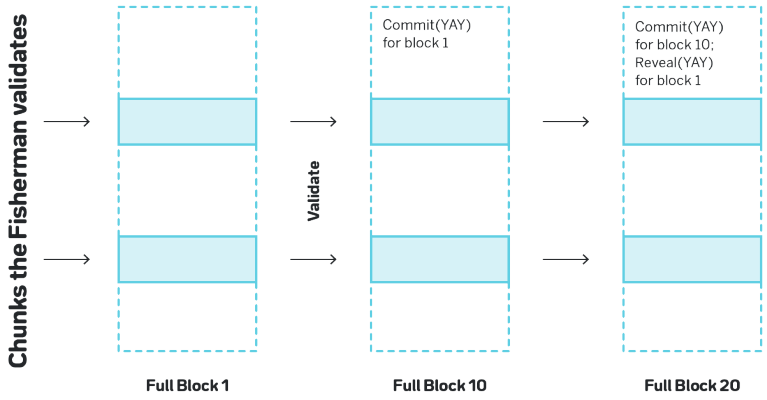

Figure 25: Commit-reveal

Instead we make the validators first commit to the validation result (either the message that attests to the validity of the chunks, or the proof of an invalid state transition), wait for a certain period, and only then reveal the actual validation result, as shown on figure 25. The commit period doesn't intersect with the reveal period, and thus a lazy validator cannot copycat honest validators. Moreover, if a dishonest validator committed to a message that attests to the validity of the assigned chunks, and at least one chunk was invalid, once it is shown that the chunk is invalid the validator cannot avoid the slashing, since, as we show in section 3.7.5, the only way to not get slashed in such a situation is to present a message that contains a proof of the invalid state transition that matches the commit.

3.7.5 Handling challenges

As discussed above, once a validator receives a block with an invalid chunk, they first prepare a proof of the invalid state transition (see section 3.7.1), then commit to such a proof (see 3.7.4), and after some period reveal the challenge. Once the revealed challenge is included in a block, the following happens:

1. All the state transitions that happened from the block containing the invalid chunk until the block in which the revealed challenge is included get nullyfied. The state before the block that includes the revealed challenge is considered to be the same as the state before the block that contained the invalid chunk.

2. Within a certain period of time each validator must reveal their bitmask of the shards they validate. Since the bitmask is created via a VRF, if they were assigned to the shard that had the invalid state transition, they cannot avoid revealing it. Any validator that fails to reveal the bitmask is assumed to be assigned to the shard.

3. Each validator that after such period is found to be assigned to the shard, that did commit to some validation result for the block containing the invalid chunk and that didn't reveal the proof of invalid state transition that corresponds to their commit is slashed.

4. Each validator gets a new shards assignment, and a new epoch is scheduled to start after some time sufficient for all the validators to download the state, as shown on figure 26.

Note that from the moment the validators reveal the shards they are assigned to until the new epoch starts the security of the system is reduced since the shards assignment is revealed. The participants of the network need to keep it in mind while using the network during such period.

3.8 Signature Aggregation

For a system with hudreds of shards to operate securely, we want to have on the order of $10,000$ or more validators. As discussed in section 3.7, we want each

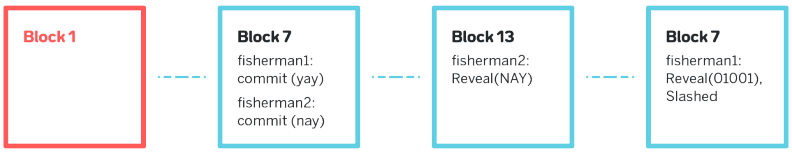

Figure 26: Handling the challenge

validator to publish a commit to a certain message and a signature on average once per block. Even if the commit messages were the same, aggregating such a BLS-signature and validating it would have been prohibitively expensive. But naturally the commit and reveal messages are not the same across validators, and thus we need some way to aggregate such messages and the signatures in a way that allows for fast validation later.

The specific approach we use is the following:

Validators joining block producers. The block producers are known some time before the epoch starts, since they need some time to download the state before the epoch starts, and unlike the validators the block producers are not concealed. Each block producer has v validator slots. Validators submit off-chain proposals to the block producers to get included as one of their v validators. If a block producer wishes to include a validator, they submit a transaction that contains the initial off-chainrequest from the validator, and the block producer's signature that makes the validator join the block producer. Note that the validators assigned to the block producers do not necessarily validate the same shards that the block producer produces chunks for. If a validator applied to join multiple block producers, only the transaction from the first block producer will succeed.

Block producers collect commits. The block producer constantly collects the commit and reveal messages from the validators. Once a certain number of such messages are accumulated, the block producer computes a merkle tree of these messages, and sends to each validator the merkle root and the merkle path to their message. The validator validates the path and signs on the merkle root. The block producer then accumulates a BLS signature on the merkle root from the validators, and publishes only the merkle root and the accumulated signature. The block producer also signs on the validity of the multisignature using a cheap ECDSA signature. If the multisignature doesn't match the merkle root submitted or the bitmask of the validators participating, it is a slashable behavior. When synchronizing the chain, a participant can choose to validate all the BLS signatures from the validators (which is extremely expensive since it involves aggregating validators public keys), or only

the ECDMA signatures from the block producers and rely on the fact that the block producer was not challenged and slashed.

Using on-chain transactions and merkle proofs for challenges. It can be noted that there's no value in revealing messages from validators if no invalid state transition was detected. Only the messages that contain the actual proofs of invalid state transition need to be revealed, and only for such messages it needs to be shown that they match the prior commit. The message needs to be revealed for two purposes:

1. To actually initiate the rollback of the chain to the moment before the invalid state transition (see section 3.7.5).

2. To prove that the validator didn't attempt to attest to the validity of the invalid chunk.

In either case we need to address two issues:

1. The actual commit was not included on chain, only the merkle root of the commit aggregated with other messages. The validator needs to use the merkle path provided by the block producer and their original commit to prove that they committed to the challenge.

2. It is possible that all the validators assigned to the shard with the invalid state transition happen to be assigned to corrupted block producers that are censoring them. To get around it we allow them to submit their reveals as a regular transaction on-chain and bypass the aggregation.

The latter is only allowed for the proofs of invalid state transition, which are extremely rare, and thus should not result in spamming the blocks.

The final issue that needs to be addressed is that the block producers can choose not to participate in messages aggregation or intentionally censor particular validators. We make it economically disadvantageous, by making the block producer reward proportional to the number of validators assigned to them. We also note that since the block producers between epochs largely intersect (since it's always the top w participants with the highest stake), the validators can largely stick to working with the same block producers, and thus reduce the risk of getting assigned to a block producer that censored them in the past.

3.9 Snapshots Chain

Since the blocks on the main chain are produced very frequently, downloading the full history might become expensive very quickly. Moreover, since every block contains a BLS signature of a large number of participants, just the aggregation of the public keys to check the signature might become prohibitively expensive as well.

Finally, since in any foreseeable future Ethereum 1.0 will likely remain one of the most used blockchains, having a meaningful way to transfer assets from

Near to Ethereum is a requirement, and today verifying BLS signatures to ensure Near blocks validity on Ethereum's side is not possible.

Each block in the Nightshade main chain can optionally contain a Schnorr multisignature on the header of the last block that included such a Schnorr multisignature. We call such blocks *snapshot* blocks. The very first block of every epoch must be a *snapshot* block. While working on such a multisignature, the block producers must also accumulate the BLS signatures of the validators on the last *snapshot* block, and aggregate them the same way as described in section 3.8.

Since the block producers set is constant throughout the epoch, validating only the first *snapshot* blocks in each epoch is sufficient assuming that at no point a large percentage of block producers and validators colluded and created a fork.

The first block of the epoch must contain information sufficient to compute the block producers and validators for the epoch.

We call the subchain of the main chain that only contains the *snapshot* blocks a *snapshot* chain.

Creating a Schnorr multisignature is an interactive process, but since we only need to perform it infrequently, any, no matter how inefficient, process will suffice. The Schnorr multisignatures can be easily validated on Ethereum, thus providing crucial primitives for a secure way of performing cross-blockchain communication.

To sync with the Near chain one only needs to download all the snapshot blocks and confirm that the Schnorr signatures are correct (optionally also verifying the individual BLS signatures of the validators), and then only syncing main chain blocks from the last snapshot block.

4 Conclusion

In this document we discussed approaches to building sharded blockchains and covered two major challenges with existing approaches, namely state validity and data availability. We then presented Nightshade, a sharding design that powers NEAR Protocol.

The design is work in progress, if you have comments, questions or feedback on this document, please go to `https://near.chat`.

References

[1] Monica Quaintance Will Martino and Stuart Popejoy. Chainweb: A proof-of-work parallel-chain architecture for massive throughput. 2018.

[2] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. *CoRR*, abs/1809.09044, 2018.

[3] Songze Li, Mingchao Yu, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Polyshard: Coded sharding achieves linearly scaling efficiency and security simultaneously. *CoRR*, abs/1809.10361, 2018.

[4] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.

[5] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[6] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.