Nightshade: Near Protocol Sharding Design 2.0

Alex Skidanov
/AlexSkidanov
alex@near.org

Illia Polosukhin
/ilblackdragon
illia@near.org

Bowen Wang
/BowenWang18
bowen@near.org

July 2019
Updated February 2024

Contents

Introduction

The past several years have seen significant progress in scaling blockchains. Back in 2018, when the early team first started to build NEAR Protocol, Ethereum was one of the only smart contract platforms available to builders and it suffered from high transaction costs and low throughput. Today, Ethereum is scaling through rollups (a.k.a. L2s[1]). Quite a few other layer 1 blockchains such as Polkadot, Solana, Sui, and Aptos that promise different kinds of scaling have launched in the past few years.

In general, most layer-one blockchains use one of two scaling approaches: vertical scaling and horizontal scaling. For vertical scaling, a few approaches are explored, including each validator operating more expensive hardware, parallelizing transaction execution, separating consensus from execution, etc. Horizontal scaling has two categories: heterogeneous and homogeneous. In the heterogeneous approach, there are different chains or execution environments that operate independently and there is usually a main chain that orchestrates everything and provides security guarantees. In the homogeneous approach, a blockchain is divided into multiple parallel shards that have the same execution environment and the protocol dictates how different shards communicate with each other. Each shard maintains its own state, which allows the blockchain to scale linearly by adding more shards.

NEAR Protocol chooses a homogeneous sharding approach to build a highly scalable blockchain. This document outlines the general approach to blockchain sharding as it exists today, 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. As of this writing of the revised version in February 2024, NEAR has been live for three and half years. There have been a total of 750M transactions and 57M accounts on chain since the mainnet launch in October 2020.

Since the original publication of the Nightshade whitepaper in July 2019, there have been a lot of new developments in blockchain protocol research, most notably zero-knowledge proofs. In the case of NEAR, during the implementation of Nightshade across different phases over three-plus years, we gained a better understanding of blockchain scalability and also came to realize some limitations of the original design. In short, we underestimated the engineering complexity of fraud proofs in a sharded blockchain and started to question whether that was still the most promising direction for the protocol. Recent advancements in stateless validation research provided an alternative path, which not only eliminates the critical dependency on fraud proofs, but also significantly increases per-shard throughput by enabling nodes to hold state in memory. As a result, we decided to update the Nightshade design to incorporate stateless validation.

This is a revised edition of the Nightshade paper, version 2.0, reflecting this change in the Nightshade roadmap. We will also elaborate on the power of zero-knowledge proofs, and how they may be combined with stateless validation to

[1]Technically a rollup is a type of L2, but the success of rollups has made the two terms almost synonymous today.

produce a future-proof sharding design that is both highly scalable and highly decentralized. New or heavily modified text is included in section 2.4, section 2.5, section 3.5, section 3.7, and section 3.8.

1 Sharding Basics

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

This design, though simple, is sufficient to outline some of the 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),

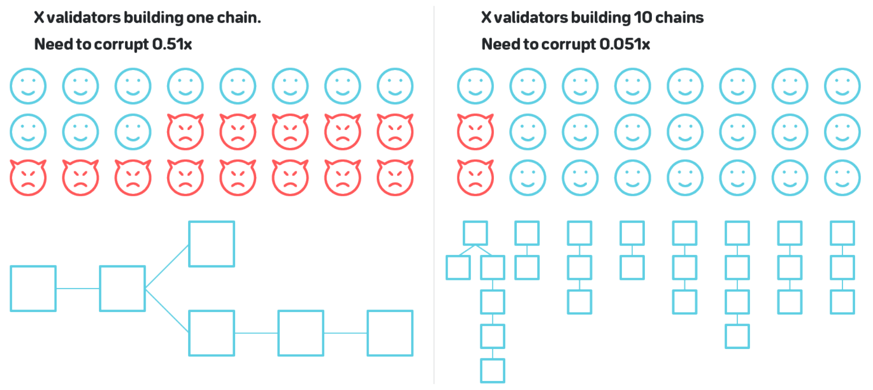
X validators building one chain.
Need to corrupt 0.51x
X validators building 10 chains
Need to corrupt 0.051x

Figure 1: Splitting the validators across shards

which brings us to the second point: who chooses the 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 a blockchain is in itself 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 validator 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 a chain as a Beacon chain. The existence of the Beacon chain brings us to the next interesting topic: 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 possible in theory to design such a sharding solution, any solution that includes 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 its operating nodes, 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 to process transactions faster.

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.

1.3 State sharding

Until now, we haven't defined clearly what exactly is and is not separated when a network is divided into shards. Specifically, nodes in a 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 task of processing transactions requires more compute power with the increasing number of transactions being processed;
2. The task of relaying transactions and blocks requires more network bandwidth with the increasing number of transactions being relayed;
3. The task of storing data requires more storage as the state grows. Importantly, unlike the needs for processing power and network bandwidth, 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 300GB — easily manageable by most of the nodes. But the number of transactions Ethereum can process per second 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 keeps 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 takes 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 bandwidth.

Practically, under State Sharding, the nodes in each shard are building their own blockchain that contains transactions affecting only the local part of the global state that is assigned to that shard. Therefore, the validators in each 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.

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

1.4 Cross-shard transactions

The sharding model we have described so far is not very useful, because if individual shards cannot communicate with each other, they are no better than multiple independent blockchains that can't efficiently communicate. The proliferation of bridges in today's multichain ecosystem is a clear signal that cross-chain (or cross-shard) communication is crucial.

For now let's only consider simple payment transactions, where each participant has an account on exactly one shard. If someone wishes to transfer money from one account to another within the same shard, the transaction can be processed entirely by the validators in that shard. If, however, Alice resides on shard #1, and wants to send money to Bob who resides on shard #2, neither the validators on shard #1 (who can't credit Bob's account) nor the validators on shard #2 (who can't 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, all blocks in the relevant shards that contain state transitions related to the transaction get produced at the same time, and the validators of the relevant shards collaborate to execute 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 has been proposed today in Cosmos, Ethereum Serenity, NEAR, Kadena, and others. A problem with this approach is 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 depicting two shards, both of which encounter 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 the 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 more

[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

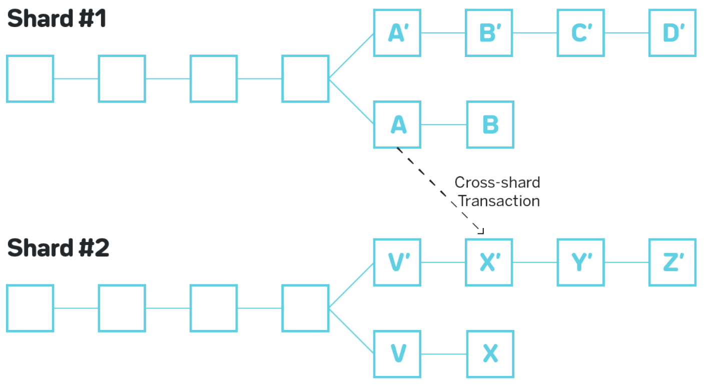

Figure 2: Asynchronous cross-shard transactions

straightforward since the block structure and consensus are the same across shards, and there's a beacon chain that can be used for coordination. However, in a sharded blockchain, all the shard chains are the same, while in the global blockchain ecosystem there are lots of different blockchains, with different target use cases, levels of 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 with a working interoperability subsystem. Such a system is unlikely to feature validator rotation, so some extra measures need to be taken to ensure security. Both Cosmos and Polkadot are effectively these types of systems.[4]

1.5 Malicious behavior

In this section we will review the adversarial behaviors malicious validators can 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 byzantine fault tolerant (BFT) or not; corrupting a sufficient number of validators will always make it possible to create a fork.

It is significantly more likely that more that 50% of a single shard will be corrupted, than that more than 50% of the entire network will be corrupted (we

[4]Refer to this writeup by Zaki Manian from Cosmos: and this Tweet-storm by the first author of this document: for a detailed comparison of the two.

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 all be 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 in general the probability of shards being corrupted is non-negligible, we can't assume that forks won't happen, even if byzantine consensus had been reached among the shard validators, or if many blocks were produced on top of the block with the state change.

This problem has multiple potential solutions, the most common 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 the shard-specific fork-choice rule for blocks 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 in figure 3.

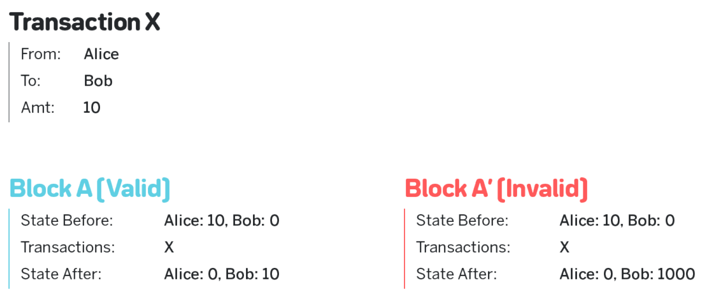
Transaction X
From: Alice
To: Bob
Amt: 10
Block A (Valid)
State Before: Alice: 10, Bob: 0
Transactions: X
State After: Alice: 0, Bob: 10
Block A' (Invalid)
State Before: Alice: 10, Bob: 0
Transactions: X
State After: Alice: 0, Bob: 1000

Figure 3: An example of an invalid block

In a classic non-sharded blockchain, such an attack is not possible, since all the participants in the network validate all the blocks. A block with such an invalid state transition will be rejected both by other block producers and non-block-producing participants in the network. Even if the malicious validators continue creating blocks on top of the invalid block faster than honest validators build the correct chain (thus making the chain with the invalid block longer than the chain without it), every participant using the blockchain for any purpose validates all the blocks, and will discard all blocks built on top of the invalid block.

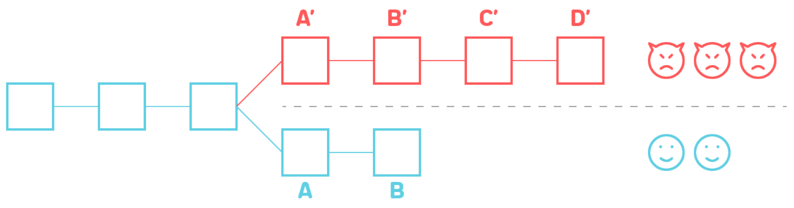

Figure 4: Attempt to create an invalid block in a non-sharded blockchain

In 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 have discarded A' as invalid and built on top 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 a classic non-sharded blockchain, every participant using the blockchain for any purpose is responsible for validating all the blocks they receive and recomputing the state. Thus any person with any interest in the blockchain would observe that A' is invalid, and thus also immediately discard B', C' and D', 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 the history of any shard of the blockchain was an invalid block included.

Note that unlike with forks, cross-linking to the beacon chain is not a sufficient solution for proving that the blockchain's history contains no invalid blocks, 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 have attested to its correctness).

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

2 State Validity and Data Availability

The core idea in sharded blockchains is that most participants operating or using the network cannot validate all blocks in all 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, a participant cannot be certain that the state they interact with is the result of some valid sequence of blocks and that this sequence of blocks is indeed the canonical chain in the shard. This is a problem that doesn't exist in non-sharded blockchains.

We will first present a simple solution to this problem that has been previously proposed by many protocols, and then analyze the way this solution can break and the attempts that 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 of 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 1/3 failing can (for practical purposes) be assumed to be zero.

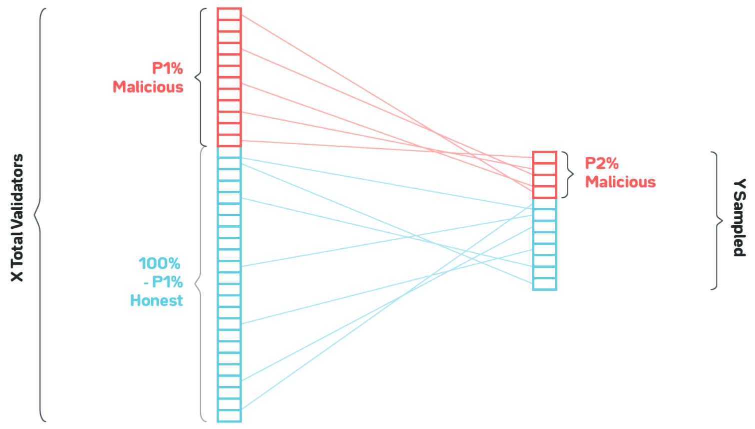

Figure 5: Sampling validators

1/3 is an important threshold. There's a family of consensus protocols, called byzantine fault tolerant (BFT) consensus protocols, that guarantees that as long as fewer than 1/3 of participants fail, either by crashing or by acting in some way that violates the protocol, consensus will be reached.

With this assumed 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 assumption is also that the current chain is the only chain in the shard. See figure 6 for a visualization.

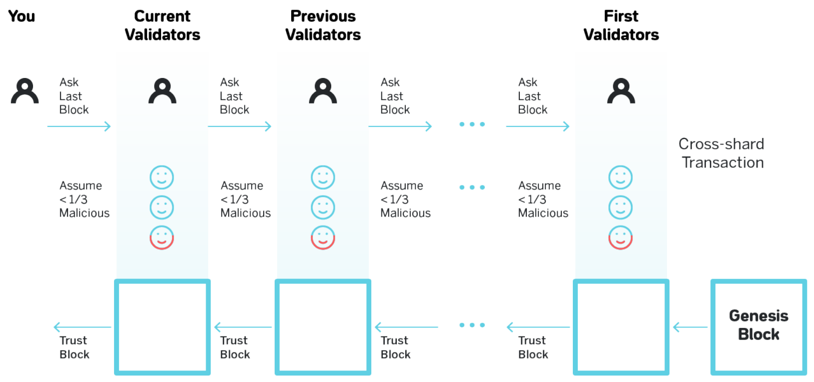

Figure 6: A blockchain with each block finalized via BFT consensus

This simple solution doesn't work if we assume that validators can be corrupted adaptively, which is not an unreasonable assumption.[5] 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 be certain about the validity of a block, we must know that at any point in history no shard in the system has had a colluding validator majority; taking into consideration adaptive adversaries, we no longer have such certainty. As we discussed in section 1.5, colluding validators can produce two basic malicious behaviors: creating forks, and producing invalid blocks.

Malicious forks can be addressed by blocks being cross-linked to the beacon chain, which 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, in which Shard #1 is corrupted and a malicious actor produces invalid block B. Suppose that in this block B 1000 tokens were minted out of thin air and deposited in Alice's account. The malicious actor then produces valid

[5] 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.

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.

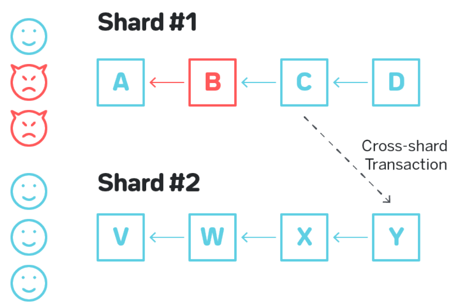

Figure 7: A cross-shard transaction from a chain that has an invalid block

Some simple approaches to tackling this problem are:

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 un-directed graph in which each shard is connected to several other shards, and only allow cross-shard transactions between neighboring shards (for example, this is how Vlad Zamfir's sharding essentially works[6], and a similar idea is used in Kadena's Chainweb [1]). If a cross-shard transaction is needed between shards that are not neighbors, such a transaction is routed through multiple shards. In this design, a validator in each shard is expected to validate all the blocks in their shard as well as all the blocks in all the neighboring shards. Consider the figure below with 10 shards, each with four neighbors, where no two shards require more than two hops for a cross-shard communication — shown on figure 8.

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

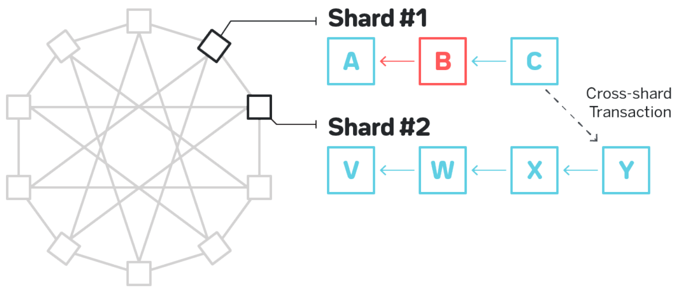

Figure 8: An invalid cross-shard transaction in a Chainweb-like system that will get detected

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

While corrupting a single shard is no longer a viable attack, corrupting a few shards remains a potential 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 by corrupting both shard #1 and shard #2.

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 a few directions towards properly solving state validity: fraud proofs, stateless validation, and cryptographic computation proofs.

2.3 Fishermen

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 is 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 much smaller than that of receiving a full block.

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

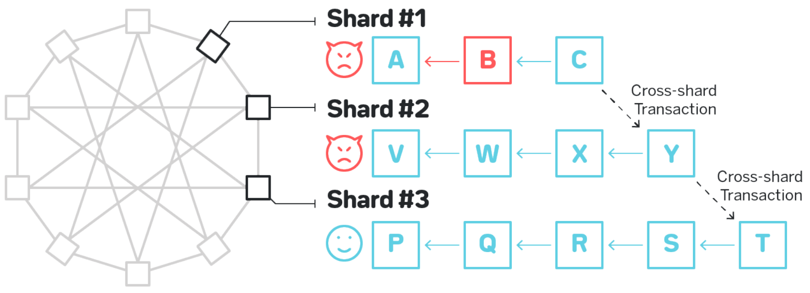

Figure 9: An invalid cross-shard transaction in a Chainweb-like system that will not get detected

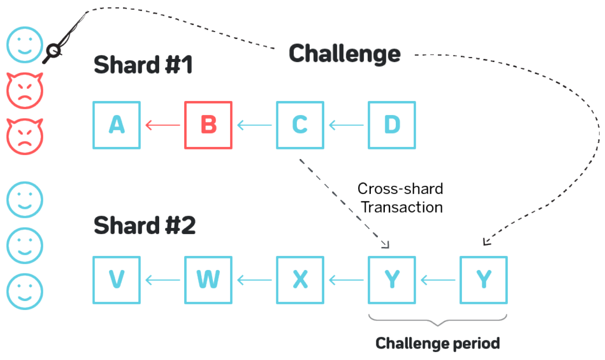

Figure 10: Fisherman

This is a well-known approach to the state validity problem. This approach, however, has a few 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 in fact invalid). Introducing such a period would significantly slow down cross-shard transactions.

2. The existence of the challenge protocol creates a new vector of attack as malicious nodes could spam the network with invalid challenges. An obvi-

ous 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 with invalid challenges (and burn their deposits) — for example, to prevent a valid challenge from an honest validator from going through. These attacks are called **Grieving Attacks**.

3. The implementation is quite complex and difficult to test. The complexity comes from two aspects: 1) making sure that a challenge can be properly processed by validators who do not have the state of the shard and 2) rolling back the state of the blockchain after a challenge is successfully processed and slashing the offending validator. The first one is not easy, as it involves building a lot of the machinery described in section 2.4. The second one is even more difficult, especially in a sharded blockchain. When an invalid state transition is found, we cannot just roll back the state of the affected shard because it is possible for an invalid state transition to mint, say, 1 billion native or fungible tokens, and send them to other shards. As a result, the state of all shards needs to be rolled back simultaneously, which leads to a multitude of problems, especially in regards to consensus and validator assignment.

   Testing challenges is even more daunting. This is a mechanism that is never expected to trigger in practice yet is crucial to the design – if someone finds a vulnerability in the implementation, it is very likely that the exploit could cause someone to lose money. This is likely the reason why after quite a few years of the idea being proposed, there is still no full (i.e., permissionless) implementation of challenges in any blockchain at the time of writing.[7]

2.4 Stateless Validation

One of the core issues mentioned in section 2.1 is the adaptive corruption of validators in a single shard. This is why the simple solution proposed above does not work. One possible way to address the issue is to rotate validators very frequently, e.g. every block. Since we assume that each validator assignment is the result of a shuffling based on the on-chain randomness beacon, adaptive corruption won't work in this case.

However, a core challenge of rotating validators with each block is determining how validators will retrieve the state to verify the validity of shard blocks, since they may be assigned to a different shard at every single block. Assuming fast blocks (˜1s) and a reasonable state size (>10GB), it is not feasible to expect that a node could download the state of a new shard and validate the new block within a few seconds.

Stateless validation provides an elegant solution. Instead of maintaining the full state to execute transactions and validate blocks, a validator is provided a

[7]Arbitrum implemented fraud proofs but as of this writing, only whitelisted entities can submit them.

state witness to validate a block. **State Witness** refers to the state touched during the execution of a chunk alonside with proof that they belong to the state as of before the chunk. More specifically, assuming that a validator needs to validate block h and it knows the prev state root s_{prev}, i.e, state root as of before block h, then a state witness is the state touched during the execution of h and the proof that they indeed belong to the state specified by s_{prev}. Using the state witness, a validator can execute and verify blocks without maintaining the state of that shard locally.

In order for stateless validation to work, some entity needs to be able to provide state witness. Usually they are part of the overall validator set so that they have an incentive to maintain the full state. It is worth noting that the state witness provider, while necessary for the network to function, could not single-handedly corrupt the network. This is because the state witness is validated against the state root and even though a single node may be responsible for producing the state witness, it has no way to produce an invalid state witness without getting caught.

This property enables a design where most validators are lightweight and only a few validators need to operate more expensive hardware. This is beneficial for the decentralization of a blockchain network as it lowers the barrier to entry to become a validator.

2.5 Succinct Non-interactive Arguments of Knowledge

Another solution to the multiple-shard corruption problem is to use a cryptographic construction that allow participants 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. zkSNARKs, zkSTARKs, 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. Despite significant progress made in this area in the past few years with the emergence of new proving systems and engineering optimizations, the state-of-the-art zero-knowledge (ZK) proving systems today such as Polygon Hermez and Risc Zero are about 10,000 times slower than native execution.

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 validating, but also 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 computing zk-SNARKs to run their processes on special hardware without reducing the decentralization of the system.

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

1. Dependence on less researched and less time-tested cryptographic primitives;

2. “Toxic waste” — zkSNARKs 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. zkSTARKs, however, do not rely on the same underlying cryptographic primitives such as KZG commitments and therefore do not suffer from the same problem;

3. Extra complexity introduced into the system design. While there are a few zkEVM rollups in production today, they rely on a single sequencer and a single prover, which greatly simplifies the design. How exactly ZK proofs should be integrated into the design of a scalable blockchain is still mostly a research topic at this point;

4. Generating zk-SNARK proofs can be computationally intensive, especially for complex computation such as hashing. The computational resources required for zk proofs inevitably lead to high cost of proof generation, which poses a barrier for wide adoption.

Despite these challenges, the field of zero-knowledge research is progressing very rapidly. When this paper was first written in 2019, few people believed that a production-grade zkEVM would be possible in the next five years. Today, however, there are multiple zkEVMs (Polygon, Scroll, zkSync, Linea, etc.) live on Ethereum mainnet. It is not unreasonable to expect that in the next one or two years, the overhead of zero-knowledge proofs would go down by one or two orders of magnitude, which would greatly improve their usability in different applications, including protocol design.

2.6 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 the parts of state and transactions they are interested in, as shown in figure 11.

Now, if a majority of full nodes collude, they can produce a block (valid or invalid) and send its hash to the light nodes, while never disclosing the full contents of the block. There are various ways they can benefit from this. For example, consider figure 12.

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

Say 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

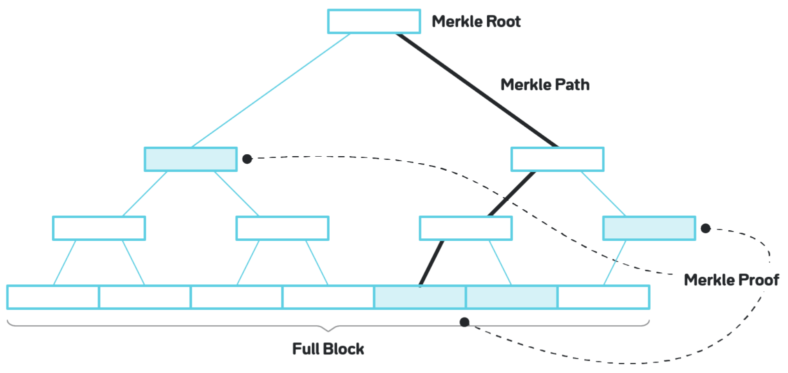

Figure 11: Merkle Tree

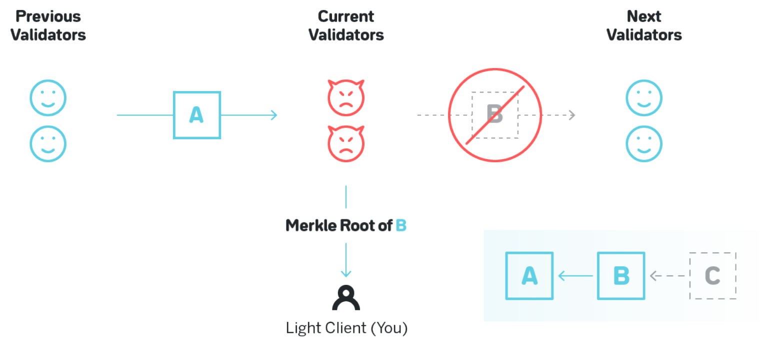

Figure 12: Data Availability problem

the money to you). Confident that the transaction is finalized, you provide the service.

However, the validators never distributed the full content of 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 — and 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 the header of an invalid block (or used it to initiate a cross-shard transaction), but never distributed the block, honest validators have no way to craft a challenge.

We will cover three complementary approaches to address this problem.

2.6.1 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 validators, who cannot be rotated as frequently, since they must download the entire state of the shard each time they rotate (as shown in figure 13).[8]

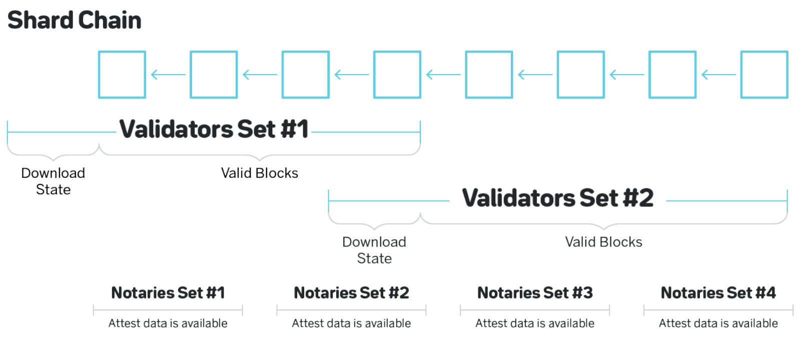

Figure 13: Validators need to download state and thus cannot be rotated frequently

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

[8]Here we assume stateless validation is not used, so validators all have to download the state of a shard.

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`.

2.6.2 Erasure Codes

When a particular light node receives a hash of a block, in order 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, because unless the light nodes collectively download the entire block, the malicious block producers can choose 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 in figure 14.

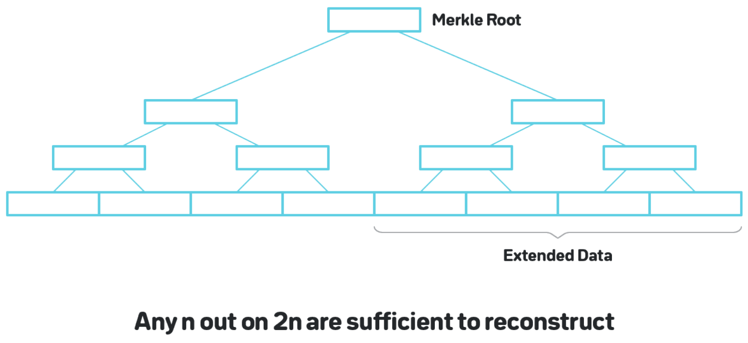

Figure 14: Merkle tree built on top of erasure coded data

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

2.6.3 Polkadot's approach to data availability

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

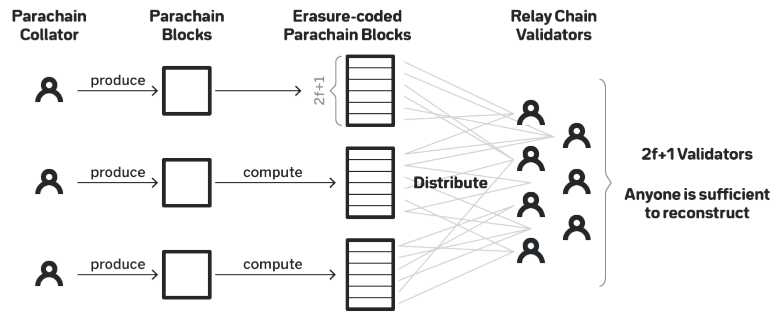

Figure 15: Polkadot's data availability

2.6.4 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 become unavailable later for a variety of reasons: nodes going offline, nodes intentionally erasing historical data, etc.

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

Long-term availability is not as pressing 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 a sharded protocol needs to be designed in such a way that the system is secure even if some of the 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 comes with certain complexities. In particular, the fork choice rule needs to be executed separately in each chain, so therefore 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 either maintains the state that corresponds to the shards that they validate transactions for, or relies on others to provide them with state to validate transactions. The list of all the transactions in the block is split into physical chunks (one chunk 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, some chunks might be missing due to network delays, so in practice each block contains either one or zero chunks per shard. See section 3.3 for details on how blocks are produced.

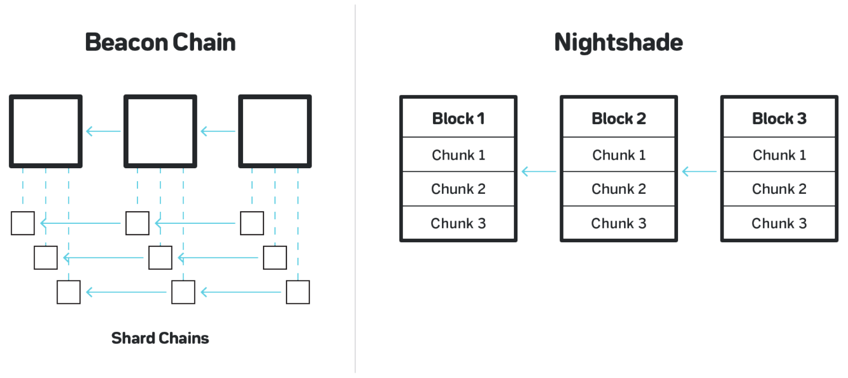

Figure 16: A model with shard chains on the left and with one chain with blocks split into chunks on the right

3.2 Consensus

The two dominant approaches to consensus in blockchains today are 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 some set of validators reach BFT consensus for each block.

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

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

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

2. It is not feasible for all network participants to participate in BFT consensus for each block; thus, usually only a randomly sampled subset of participants reaches consensus. A randomly sampled set can, in principle, be adaptively corrupted, and in theory a fork 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 in addition to BFT consensus, or should 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, a temporary network glitch or a network split can completely stall the system. The system ideally should be able to continue to operate, as long as at least half of the participants are online (heaviest chain-based protocols continue operating even if fewer than half of the participants are online, but the desirability of this property is debated within the blockchain community).

A hybrid model, in which the consensus used is some sort of heaviest chain approach, but some blocks are periodically finalized using a BFT finality gadget, maintains the advantages of both models. Such BFT finality gadgets are Casper FFG [6], used in Ethereum 2.0[9], Casper CBC, and GRANDPA used in Polkadot.

Nightshade uses 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. The weight 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 heaviest-chain consensus, Nightshade uses a finality gadget that uses 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. Instead the gadget only introduces extra slashing conditions such that once a

[9]Also see the whiteboard session with Justin Drake for an in-depth overview of Casper FFG, and how it is integrated with GHOST heaviest-chain consensus.

block is finalized by the finality gadget, a fork is impossible, unless a very large percentage of the total stake is slashed. The full details of NEAR's consensus algorithm can be found in the Doomslug paper.

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, does have a separate stake.

The system contains n shards, where $n = 100$ 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, which we refer to as the *main chain*. The state of the main chain is split into n shards, and each block producer has at any moment only downloaded locally a subset of the state that corresponds to some subset of the shards, and only processes and validates transactions that affect those parts of the state. Validators do not maintain the state of any shard locally, but they do download and verify all the block headers. They validate chunks using state witness created by chunk producers. The details of this mechanism will be discussed in section 3.5.

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 roughly 16 hours. 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 $s_w = 4$, 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 shard s, one of the assigned block producers to s 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, and the Merkle root of the final state. When a block producer produces a chunk, they also produce an associated state witness required to execute the chunk.

Throughout the rest of this document we refer to the block producer that is responsible for producing a chunk at a particular time for a particular shard as a *chunk producer*. A chunk producer is always a block producer and vice versa, this distinction is made to help provide more accurate context when we discuss block production or chunk production.

Within an epoch, the block and chunk production schedule is determined by a randomness seed generated at the beginning of the epoch and for each block height, there is an assigned block producer. Similarly, for each height, there is an assigned chunk producer for each shard.

Since chunk production, unlike 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 are responsible for producing chunks and associated state witnesses.

3.4 Ensuring data availability

To ensure data availability, Nightshade uses an approach similar to that of Polkadot described in section 2.6.3. 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 of the block producers building it can have unavailable chunks.

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

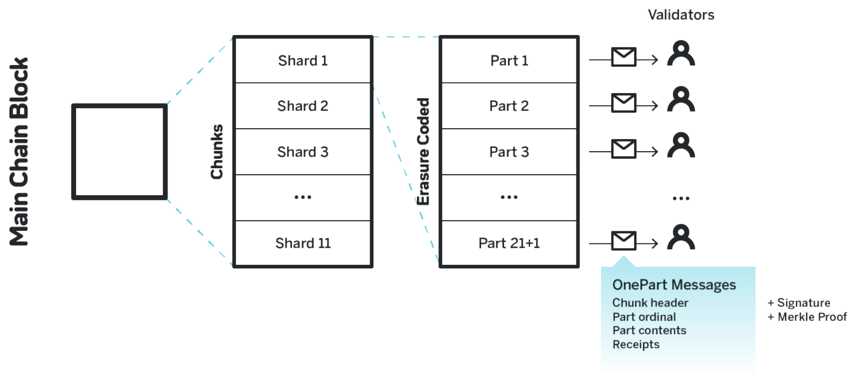

Figure 17: Each block contains one or zero chunks per shard, and each chunk is erasure coded. Each part of the erasure coded chunk is sent to a designated block producer via a special *onepart* message

producer since it is impossible to prove later that the block producer didn't have the *onepart* message.

To address the lazy block producer problem, each chunk producer when creating a chunk must choose a color (red or blue) for each part of the future encoded chunk, as well as 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, 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 blindly sign the block.

3.5 State transition application

A chunk producer chooses transactions to include in a chunk, applies the state transition, and generates state witness along the way when they produce a chunk. More specifically, when a chunk producer applies a chunk they produces, they record all the trie nodes visited during the execution. This set of trie nodes is the state witness for this chunk. The chunk header contains a Merkle root of the Merkleized state after the chunk is applied.[10] The state witness is then shared with validators assigned to this shard for the next block so that they could validate the chunk.

A validator only processes a block if:

1. The previous block was received and processed;
2. For each chunk, it receives the *onepart* message if it is not assigned to the corresponding shard.
3. For each chunk, it receives the full chunk if it is assigned to the corresponding shard

Once the block is being processed, a validator applies the state transition of the shard they are assigned to and checks whether the resulting state root matches what is posted in the chunk header. If they match, the validator sends an *chunk endorsement* to the next block producer. Block producers would only include a chunk in a block if they have more than 2/3 endorsements from validators assigned to the corresponding shard.

3.5.1 State Representation

In section 2.4, we define state witness as the state touched during the execution of a chunk, along with a proof that the recorded state pieces indeed belong to the state as specified by a state root. An important question left unanswered is how the state is represented. In blockchains where state proofs are required, their state is usually represented as a Merkle Patricia Trie (MPT). However, even for MPTs, the branching factor matters for the size of state witness — a critical component of the stateless validation design. A branching factor of 16[11] means that when the state is full, each branch node will have an overhead of 15 hashes, which is 480 bytes. In comparison, a branching factor of 2, or binary Merkle trees, only have an overhead of 1 hash, which is 32 bytes, at each branch node. An analysis shows that based on Ethereum data, the state witness size can be reduced 40% by switching the state representation to a binary MPT.

Recent research on Verkle Trees shows even more promising results. Verkle trees only require one 48-byte KZG commitment [7] per level. Therefore Verkle trees can have a large branching factor (256 - 1024), which results in much shallower trees and smaller witness sizes. Verkle tree is currently considered to be the best state representation for stateless validation.

[10] This is not exactly what is implemented. To allow for an easier upgrade from the previous version where the previous state root instead of the post state root is stored in the chunk headers, the implementation is changed to accommodate that. We describe the design conceptually here to make it easier for readers to understand.

[11] This is what Ethereum and many other blockchains use as of this writing.

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, which 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 should be 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 within block B, they fetch all the receipts generated by applying the transactions from block A for shard a, and include them in the chunk for shard a in block B. Once such a chunk is generated, cp_a produces its erasure coded version and all the corresponding *onepart* messages. cp_a knows which block producers maintain the full state for which shards. For a particular block producer, bp, let S_{bp} denote the set of shards that bp maintains full state for. Let $\mathbf{R}$ denote the receipts that are results of applying transactions in block A for shard a that have any shard in S_{bp} as the destination shard. Then cp_a would include $\mathbf{R}$ in the *onepart* message when they distribute the chunk for shard in block B to bp.

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 participant 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 for which the participant maintains state as their destination. When applying the state transition for a particular shard, the participant applies 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.

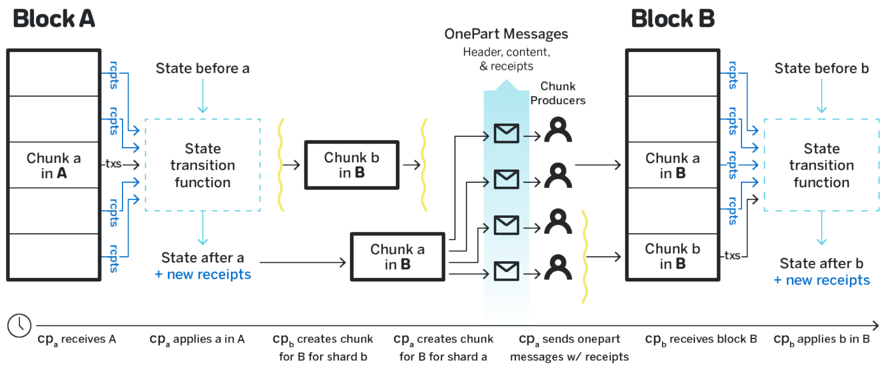

Figure 18: The lifetime of a receipt transaction

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.

To address it we use a technique similar to that used in QuarkChain [12]. 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 an imbalanced load remains for a long time, the delay from the receipt creation until the application can continue growing indefinitely. One way to address this is to drop any transaction that creates a receipt targeting a shard that has a processing delay exceeding some constant (e.g. one epoch).

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

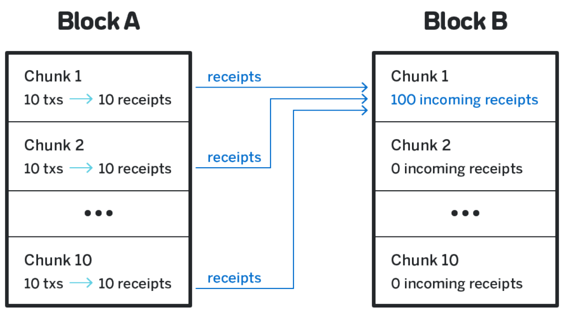

Figure 19: If all the receipts target the same shard, the shard may not have the capacity to process them

Consider figure 20. By block B the shard 4 cannot process all the receipts, so it only processes receipts originating from up to shard 3 in block A, and records this. 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.

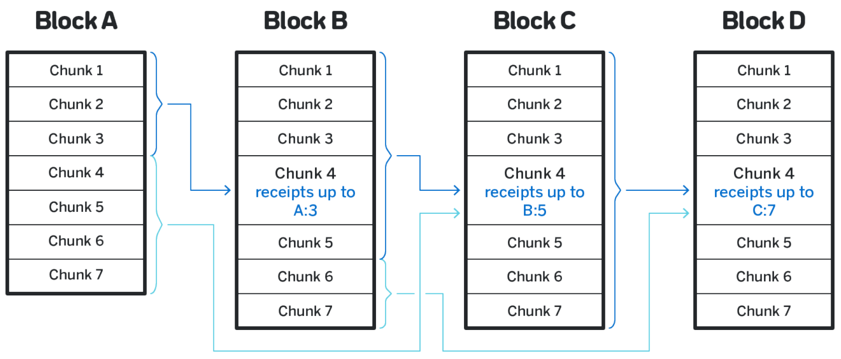

Figure 20: Delayed receipts processing

3.7 Chunks validation

As mentioned in section 3.5, a chunk is validated by a set of stateless validators assigned to the shard. In this section, we provide a more detailed analysis of the security of the design. Again, we assume the existence of a on-chain randomness beacon that can be used to generate random seeds with each block, which then can be used to shuffle validators.

The core problem here can be formulated as follows: let's assume a total of n validators (with equal stake) where at most 1/3 of them are malicious. There are s shards and k validators are assigned to each shard at every block, so $n = sk$. What is the probability of a random assignment of validators to a shard that results in a shard getting corrupted?

First, it is worth noting that a shard being corrupted means that more than 2/3 of validators assigned to that specific shard are malicious and can perform an invalid state transition. If less than 2/3 of validators are malicious, it is still possible for the chunk to be skipped due to insufficient endorsement, but no invalid state transition is possible. In addition, since validators rotate every single block, it is not possible for a shard to get indefinitely stalled due to a bad assignment.

Now let's consider the probability of a single shard getting corrupted. Let p denote the probability of a randomly chosen validator being malicious (so $p \leq 1/3$). For a single shard to have l malicious validators, the probability is hypergeometric distribution:

$$P(X = l) = \frac{\binom{k}{l}\binom{n-k}{k-l}}{\binom{n}{k}}$$

Therefore, the probability of having more than 2/3 malicious validators for a single shard is

$$P(X \geq 2k/3) \leq e^{-D(p+1/3||p)k} = e^{-\frac{k}{3}}$$

by Chernoff bound.

For s shards, the probability that at least one shard is corrupted is

$$P_{bad} \leq \sum_{i=1}^{s} P(\text{one shard is corrupted}) \leq se^{-\frac{n}{3s}}$$

When n and k are large, i.e, there are many validators assigned to the same shard, the last term $se^{-\frac{n}{3s}}$ can be made negligbly small. Our numeric calculation based on multivariate hypergeometric distribution shows that with 800 validators and 4 shards, the probability of the networking getting corrupted is roughtly 10^{-29}, which means that in expectation it takes 10^{29} blocks for the system to fail. Assuming one block per second, that translates to 3×10^{21} years. Therefore, while the security of the system is probabilistic, it is safe enough in practice.

3.7.1 Zero-knowledge proofs

One problem that is often ignored in blockchain design is what happens when the BFT assumption is violated, i.e, if there are more than 1/3 malicious validators in a blockchain. Obviously the consensus could be broken and the network may fail to produce blocks. More importantly, the malicious validators could potentially attempt to create an invalid state transition. In a non-sharded blockchain, doing this would not only require a higher total stake (2/3 of validators being malicious), but also it is also very likely to be immediately noticed and a social consensus may ensue to resolve the attack. This is because the likelihood of every single validator being corrupted in a non-sharded blockchain is extremely low; there may also be others running RPC nodes who would notice the invalid state transition, as well.

In a sharded blockchain, however, things are a bit different. It is more likely that the validators assigned to a shard could all be malicious and it is less likely that many people would run nodes that track every shard. This gives rise to the theoretical possibility of an invalid state transition attack occurring with few or any participants noticing. The result would be disastrous: a billion tokens could be minted out of thin air and moved to other shards and potentially even be bridged to other blockchains.

This is where zero-knowledge proofs (ZKPs) can come in. Since ZKPs are very cheap to verify, if there is a proof generated for the state transition of each shard, then a very large number of people can independently verify the state transition without having to rely on the signatures of validators to trust the validity of a state transition. More specifically, every wallet could run a ZK full client that verifies the state transition of the entire blockchain and be assured that no invalid state transition has happened in the history of the blockchain.

ZKPs also enable a stateless validation design that is simpler and more powerful than the one described in section 3.7. Instead of validators executing the chunk using state witness and verifying the state transition, they could verify one ZKP instead – which is much cheaper. The problem, however, is that generating a ZKP takes a long time. So if we naively swap out state witness with a ZKP, it won't work due to the proof generation latency.

Recent advancements in separating consensus and execution [8] provide a possible solution: blocks and chunks can be produced and optimistically executed while ZK proofs are being generated. Once a ZKP is generated, it is submitted into a block for all validators to verify. Because ZKPs can be small in size and quite cheap to verify[13], they can be included in the blocks directly and all validators can validate the state transition of all shards[14]. In this design, the validators would be much more lightweight, given that they only need to verify some ZKP. This means that the cost of operating a validator is going to be much cheaper and that the validator set could be much larger, which further improves the decentralization of the design.

[13]While STARK proofs can be relatively large (a few hundred kilobytes), they can be compressed again using a SNARK which would result in a proof of a few hundred bytes.

[14]Alternatively, proofs from different shards can be aggregated into one ZKP for a block.

3.8 State Size

Sharding allows us to divide the state of the entire blockchain into those of individual shards, which provides a solution to the state growth problem that is becoming more and more imminent today. The stateless validation approach makes the blockchain even more performant: there are relatively few chunk producers and they can afford to operate more expensive hardware, which means that for each individual shard, the state of the shard can be held in memory when chunk producers produce and process chunks. Validators, on the other hand, receive state witness, which is small in size, when they need to apply a chunk and can also have the required state in memory. As a result, the state reads and writes are very fast.

The cost of operating a chunk producer is not going to be exorbitantly high either. If we limit the size of each shard to 50GB, then a chunk producer could be operated with a 64GB RAM machine, which is fairly commonplace today. This asymmetry between chunk producer and validator also works well when state witness is replaced by ZKPs. Chunk producer could also operate a prover, which requires more hardware resources and the validators can run on even cheaper hardware due to the low cost of verifying ZKPs.

3.9 Snapshots Chain

Since the blocks on the main chain are produced very frequently, downloading the full history might become expensive very quickly. 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. Such a chain can also be built and validated on Ethereum inside a smart contract can provides a secure method of cross-chain communication.

To sync with the NEAR chain, one only needs to download all the *snapshot* blocks and confirm that the signatures are correct, and then only have to sync the 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 that come with existing approaches, namely state validity and data availability. We then presented Nightshade, a sharding design that powers NEAR Protocol.

The design is a work in progress. If you have comments, questions, or feedback on this document, please go to `https://github.com/near/neps`.

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.

[7] Aniket Kate, Gregory Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. pages 177–194, 12 2010.

[8] George Danezis, Eleftherios Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient BFT consensus. *CoRR*, abs/2105.11827, 2021.