Halting the Solana Blockchain with Epsilon Stake

Quentin Kniep
ETH Zurich
Switzerland
qkniep@ethz.ch

Fabian Schaich
ETH Zurich
Switzerland
schaicfa@ethz.ch

Jakub Sliwinski
ETH Zurich
Switzerland
jsliwinski@ethz.ch

Roger Wattenhofer
ETH Zurich
Switzerland
wattenhofer@ethz.ch

ABSTRACT

Solana is a blockchain protocol that has gained significant attention in the cryptocurrency community. This work examines Solana's consensus protocol and its reference implementation.

In this paper we try to get an understanding of the Solana protocol. However, this is not so easy because the publicly available resources are insufficient to specify the details of the protocol. Moreover, the implementation has deviated in undocumented ways from the available protocol design description. Thus the consensus rules and their implied security properties remain unclear.

We evaluate a number of experimental scenarios in a local Solana testnet. These tests seem to confirm our basic understanding that Solana does not fully achieve consensus. In this paper we show how a single malicious validator, once elected as leader, might be able to halt the Solana blockchain. We also observe some inconsistent behavior, which is not readily explained by any of the consensus rules we are aware of.

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security*.

ACM Reference Format:
Quentin Kniep, Fabian Schaich, Jakub Sliwinski, and Roger Wattenhofer. 2024. Halting the Solana Blockchain with Epsilon Stake. In *25th International Conference on Distributed Computing and Networking (ICDCN '24), January 4–7, 2024, Chennai, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3631461.3631553

1 INTRODUCTION

Solana [19] is one of the largest blockchain systems by many metrics, including active addresses, daily transactions, and the multi-billion dollar market capitalization of its native SOL token.[1] Its popularity and valuation certainly suggest widespread trust by users, developers and investors alike.

[1]https://app.artemis.xyz/comparables

So far there has however been virtually no academic research analyzing the underlying consensus protocol. This is in contrast to other popular blockchains, e.g., Bitcoin [11], Ethereum [16], or Algorand [8], which have received a lot of scrutiny from academic researchers. The protocol is based on a novel mechanism for Sybil resistance, which the authors call *proof-of-history (PoH)*. It can be considered a hybrid between the widely-adopted proof-of-stake and proof-of-work mechanisms. At its core is a hash chain that is maintained by all validators in order to verifiably split time into slots for leader election.

Interestingly, the project made news headlines multiple times with long outages of the entire blockchain, most of which required manual intervention to get the blockchain running again.[2] Are these events expected hiccups in the ongoing development process, or do they suggest something more fundamental about the properties of the underlying consensus protocol?

In a recent write-up, Victor Shoup [14] doubts the advantages of proof-of-history. Here, however, we will focus on the consensus rules in place to achieve consistency and agreement, such as the fork choice rules for resolving temporary inconsistencies.

Our Contribution

Our main goal is to understand and analyze the Solana protocol and its consensus rules. For understanding the architecture, interpreting the experimental outcomes, and to inform the setup of our experiments, we rely on sources including not only the official whitepaper [19], but also the official documentation[3] and reference implementation[4] of the project. Finally, we also relied on discussions between project maintainers and validator administrators from the project's official communities on Reddit[5] and Discord.[6]

We experimentally evaluate interesting scenarios based on our understanding of Solana's consensus rules in a local testnet setup, which uses their official Rust SDK. The implementation of our custom scenarios is constructed based on other tests written by Solana core developers and uses an internal testing framework from their open source codebase. Specifically, we explore scenarios where a single malicious leader introduces multiple forks and analyze whether the honest validators can recover and reach consensus.

[2]https://status.solana.com/history
[3]https://docs.solana.com
[4]https://github.com/solana-labs/solana
[5]https://www.reddit.com/r/solana
[6]https://discord.com/invite/kBbATFA7PW

Our results confirm our suspicions that there are scenarios where a fork remains permanent. They also show some strong similarities to real-world outages of the Solana mainnet, including one which happened during our work on this project [15]. We thus identify scenarios in which an adversary with almost no stake may be able to, once elected as leader, cause the validators of the network to diverge, creating a permanent chain fork. Further, we identify scenarios where we cannot explain the behavior of the validators based on our understanding of the consensus rules.

2 PRELIMINARIES

In this section, we take a look at the key components of the Solana architecture.

2.1 Proof-of-History

The concept of proof-of-history uses a *hash chain* to introduce a global time measurement. This hash chain is created by continually using the output of the previous hash iteration as input to a cryptographic hash function. Because the hash rate is limited by the fastest available hardware, there is a global upper bound to the hash rate. It is also possible for a node to include additional information, such as another hash of data, to prove that this data was known before the corresponding hash chain element. This hash chain can be used by any validator locally as an approximation of a global clock. Thus, every validator knows approximately when it is their time to produce blocks according to the schedule, and they can prove to other validators that at least the required amount of time has passed by providing (part of) the hash chain.

Unfortunately, the local hashing is only an imprecise measure of time since the hash rates will vary from validator to validator and over time. This is due to differing hardware and software between validators and general inaccuracies introduced by scheduling of other processes and threads on the machine at the same time, especially during periods of high load. Furthermore, it introduces additional power consumption as every validator needs to be hashing all the time [14]. The latter is rather ironic if one thinks about it as combining the proof-of-stake mechanism and its supposedly low power consumption with a mechanism that forces all nodes to be hashing constantly to keep their clock up to date, which almost resembles a proof-of-work system.

2.2 Time Units in Solana

The time in a network is broken down into several units, starting at the smallest time unit we have:

- **Hash**: Solana's system parameters are set with the expectation that the hash rate of each network participant is around 2,000,000 hashes per second, which was measured to be the hash rate of an "Intel Xeon e5-2520 v4".[7]
- **Tick/Slot**: One tick is 12,500 hashes and serves as a timestamp entry in a ledger. At the assumed network hash rate this happens every 400 ms. This also is the time window in which a block should be produced, i.e., for which transactions need to be considered and put in a block by the current leader (according to the leader schedule).
- **Epoch**: Based on the same assumption on hash rate, an epoch takes about two days (it is defined as 432,000 slots).

[7] https://github.com/solana-labs/solana/blob/master/sdk/program/src/clock.rs

2.3 Validators

A node actively participating in the network with voting power proportional to its staked funds is called a *validator*. Validators are responsible for forwarding any received user transactions to the current leader, i.e., the block producer of the current slot, and for voting on valid blocks submitted by this leader.

Voting power is proportional to the stake that is delegated to a validator. A validator may also abstain from voting during a slot if (a) no block was received, (b) a *lockout timer* is pending, or (c) it disagrees with the current block as it is on another fork. While a validator does not receive any information from the current leader, it has to continue its hash chain as it needs to prove to other validators that in the time passed it spent time waiting for the information. To incentivize honest behavior, every vote on a fork is connected with a commitment which is represented by a *lockout timer* regarding votes on other forks. The timers are stored in a stack data structure and prevent the validator from voting on another fork for the specified duration. Initially, when a validator votes on a fork, a lockout timer of duration 2 slots is pushed to the stack. Every lockout timer already present in the stack with equal duration, i.e., duration 2, gets doubled. This step is considered after each doubling, meaning that if a lockout timer gets doubled to a duration of 4 slots and there is already a lockout timer with a duration of 4 slots the latter gets doubled as well, and so on. When a lockout timer expires, a validator then, if compliant with other timers, can vote on another fork starting from the block which was being voted on. Thus, if a validator wishes to switch forks because they, e.g., observe a majority switching forks, it can abstain from voting to decrease its timers and not introduce new ones. If a voted block lies more than 32 votes back, i.e., the lockout timer on this block would be 2^{32} slots, the block is considered finalized by the validator, i.e., is never being rolled back. This and the vote of a supermajority ($> \frac{2}{3}$ stake) on a specific slot are the only two ways to finalize a block. In addition to forwarding transactions, validating and voting, validators should also pass messages of events in the network such as new nodes joining.

2.4 Leader

A *Leader* is a validator whose task is to produce blocks for certain slots. The slots are assigned at every epoch start in a so-called *leader schedule*. When defining a new leader schedule validators get placed in a weighted index according to their stake. The individual stake placed in the index represents the probability of being sampled with respect to the sum of the stakes. So, for each slot in the leader schedule the probability that validator $i \in \{1, 2, \ldots, n\}$ with stake_i will be selected as leader is:

$$\Pr[i] = \frac{\text{stake}_i}{\sum_{j=1}^{n} \text{stake}_j}$$

where n is the total number of validators in the active set and $stake_1, \ldots, stake_n$ are their respective stake amounts.

Simply being a validator with non-zero stake is not sufficient to have a probability of being picked in the schedule as only validators are considered which are in the *active set*. Only validators who have voted at least once in a cluster-defined time window are placed in the active set.

When selected as a leader in the schedule, a validator is in charge to produce blocks for a cluster-defined number of consecutive slots (mainnet: 4 slots). An entry of a block built by a leader contains the following information:

- A unique ID which is the hash of the Entry before it.
- The hash of the transactions within it.
- The number of hashes performed since the previous entry.

3 CONSENSUS RULES

In this section we explain Solana's consensus protocol: First as it is explained in the original whitepaper and then how we understand it based on the other resources and our experiments in Section 4.

3.1 Claims from Solana Whitepaper

In 2018, Yakovenko released the Solana whitepaper [19]. It introduces a new consensus mechanism called *Proof-of-History*, which is a variant of Proof-of-Stake and describes Solana, a blockchain architecture based on this mechanism. The most important claims are summarized below with additional comments if needed due to more recent adaptations of the real implementation:

- Proof-of-History combined with Proof-of-Stake can reduce messaging overhead to "[...] sub-second finality times".
- Proof-of-History provides a verifiable passage of time, i.e., serves as a *verifiable delay function* (VDF) [2]. This allows for every node in the network to rely on the recorded passage of time without trust.
  - Although Proof-of-History provides a rough measurement for the passage of time, it is technically not a VDF because verification requires the same amount of computation as it took to create the proof and is only faster when performed in parallel [14].
- "In terms of CAP theorem, Consistency is almost always picked over Availability in an event of a Partition".
- Proof-of-History outputs can be re-computed and efficiently verified by the other validators in parallel.
- Proof-of-History sequences can be extended with additional inputs, e.g., hashes of other data, to serve as a timestamp.
- When using a collision resistant hash function with appended data it should be computationally impossible to precompute any future values of the hash chain even with prior knowledge of the events (but without their exact time of insertion).
- Sent messages are signed by the individual validator or leader.
- Proof-of-History "[...] provides some protection" against long-range attacks by making it hard to produce a historical record signed by an old private key. In the end, this would require an attacker to have "[...] access to a faster processor than the network is currently using".
- New Proof-of-History generators, i.e., leaders, get voted and selected on special occasions such as forks, runtime exceptions, and network timeouts.
  - The concept of Proof-of-History generator got replaced by leaders and their leader schedule which gets created every epoch as described in Section 2.4.
- Nothing-at-stake problem is solved by *slashing* validators, i.e., taking (part of) their staked tokens. However, to this date, slashing has not been implemented, and the current implementation ensures that "[...] after a safety violation, the network will halt" [7].
- To detect inattentive validators, invalid hashes should be randomly injected to slash voting validators. Since slashing itself has not been introduced yet, this feature has also not yet been implemented.

3.2 Our Interpretation

The most important rules for consensus, which followed from our experiment scenarios in Section 4 are:

- A block is finalized when (a) it has received votes on at least 32 slots later on the same fork, or (b) any following slot on the same fork receives a supermajority ($> \frac{2}{3}$) of votes.
- A competing fork is only considered for voting if it has $> 38\%$ stake (this is called *switch fork threshold*).
- A duplicate version of a block (created by a rogue validator) needs $> 52\%$ stake to be considered for voting (this is called *duplicate threshold*).
- There seems to be an additional threshold at 61% (as mentioned in Section 4.2) for which three validators decide on a middle validator's fork if it has >61% stake. However, we did not find a constant with this value in the implementation.

The switch fork and duplicate threshold follow from an accepted proposal on *Leader Duplicate Block Slashing* which does not actually implement slashing but lists several rules on when slashing should be applied.[8] (Note: Although in the proposal it says that the duplicate threshold is the "[...] minimum percentage of stake that needs to vote on a fork with version X of a duplicate slot, in order for that fork to become votable." the implementation uses the threshold as an upper threshold for the case to ignore the fork, i.e., $\leq 52\%$ stake results in ignoring the fork. This already raises a problem, since this threshold of 52% stake probably works if only two duplicates occur but lacks functionality for more duplicates. With the risk of Byzantine nodes, no liveness can be guaranteed which leads to a violation regarding consensus. Even worse is that sharing duplicates can be done by a single leader in its leader schedule making it even easier to act maliciously.

Possibly, a leader could share multiple blocks in the same slot. If we now divide the honest validator set into groups of $\leq 38\%$ stake each and let the group see pairwise different versions of the block for the same slot before letting them see the others, the network would stall since none of the forks reaches a switch fork threshold, meaning none of them gets considered as a valid fork once the duplicate blocks are being observed.

An adversary could force such a scenario to happen by just having their validator selected as leader for a single slot. Since Solana does not seem to have a minimum stake requirement for validators, this would allow denial-of-service attacks on the whole network by an adversary with just epsilon stake.

[8]https://github.com/solana-labs/solana/pull/16127

4 EXPERIMENTS

For testing and simulating the implementation behavior, the module *LocalCluster* was used, which is part of the Solana Rust SDK.[9] It allows setting up a local test network with different validators and is mainly used by existing test cases in the Solana codebase for benchmarking and observing voting behavior. To set up a local cluster, a `ClusterConfig` struct is needed, which is defined in the same module. In the tests, the cluster configuration was mostly set up identically to internal integration tests, with the following properties explicitly being set:

- `validator_configs`: The individual configuration per validator. This mostly remained the default implementation, except for a modified leader schedule, which we explicitly set for all validators according to each scenario.
- `validator_keys`: A list of tuples containing the key pair of a validator and a boolean indicating its inclusion in the genesis block. The latter of which is always set to true.
- `node_stakes`: The stake of all validators in the network. This is calculated by: test case stake $\cdot$ 10^{10} Lamports, i.e., 10 SOL.
- `ticks_per_slot`: The number of ticks per slot. Implicitly defines the number of hashes per slot. This value was changed according to the tested scenario and varied between 512–2048 slots (default is 64 slots).
- `skip_warmup_slots`: Whether to skip the usually required "warm up" period before allocated stakes of validators become active. This was always set to true (the default behavior for integration tests).

All the other attributes of the struct were left to their default values. To set up a common state before observing behavior, the tests relied on copying entire ledgers to other validators and resetting their votes to let them build their state with a given block store. Hence, after running individual validators first to a specific slot number and copying their state according to a scenario, all the tests then continued by restarting all relevant validators in the local cluster and observing their behavior. In order to keep track of the states, each block store of each validator is saved to a text document every time a new vote is observed or after a pre-defined time interval has passed. There is a limit after which a test case got aborted if no new votes were seen (default: 30 s). These documents then allow to analyze the different states of the validators after a specified number of slots while running the test cases.

All experiments were conducted on a commit that was released on the 19th of September 2022.[10] The test cases were run on a personal computer with an Intel Core i7-8700 and 25 GB of RAM.

4.1 Duplicate Slot

In this scenario the behavior of a cluster of three validators receiving three different versions of a block for the same slot, i.e., introducing three duplicates was observed.

A validator with epsilon stake, hereinafter referred to as *dummy* validator, was used, which was in charge of producing blocks for the first four slots. In addition, three more validators were used on which the resulting voting behavior was observed, by copying an individual state from the dummy. To reduce the number of moving parts in our experiments the leader schedule was fixed beforehand, as shown in Table 1.

[9]https://github.com/solana-labs/solana/tree/master/sdk
[10]https://github.com/solana-labs/solana/commit/9f097301df4395557e9d84212d8ddf2d4c703619

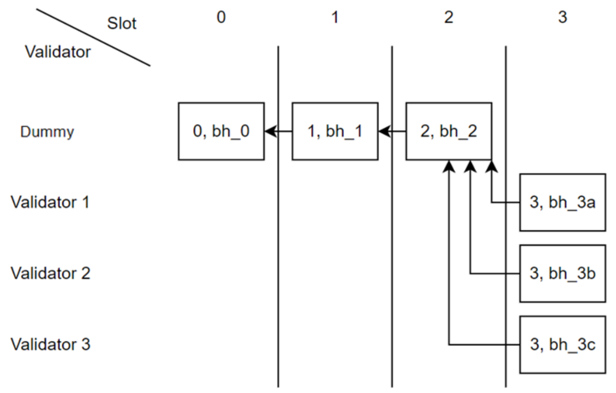

Figure 1: Initial state of the *Duplicate Slot* scenario. The blocks in slot 3 are three conflicting versions of the same block, all produced by correct leader (the dummy validator).

Slots	0–3	4–7	8–11	12–15	16–19	···	48-51
Leader	D	1	2	3	1	···	3

Table 1: Leader schedule for the *Duplicate Slot* scenario, where D indicates the dummy validator.

The dummy validator was started and left running until its block for slot 3 was observed. Afterward, all blocks in slots > 3 were purged before copying the complete ledger to the first validator without the votes of the Dummy. Then this step was repeated for the second and third validators by removing the block in slot 3 and all votes of the Dummy and letting it vote and produce a new block with a different block hash for slot 3 again. After the Dummy produced all three duplicates for slot 3, it is terminated and no longer participates in voting. This leaves us with an initial state for the validators as shown in Figure 1.

We divide the results of this analysis into the different experiments from Table 2 with their varying stake distributions.

- **Test 1** ($33.\bar{3}\%, 33.\bar{3}\%, 33.\bar{3}\%$)**:**
  Although a case with exactly equal stakes is unrealistic, it can answer the question whether there are additional consensus rules involved when the mechanism cannot rely on stake majorities. Furthermore, in this case, all forks are under the duplicate threshold, meaning voting on another fork should not be possible once the duplicate is noticed. Running this experiment, we always make the same observation: Validator 1 votes up to slot 7 while validators 2 and 3 stop voting at slot 3.
  This outcome can be interpreted in the following way:
  - Validator 1: Since validator 1 is the first to be restarted, it does not immediately see the duplicate block version in slot 3 and continues on its schedule range from slot 4 to 7 by directly appending blocks on its version of the block in slot 3.

– Validators 2 and 3: cannot verify the blocks of validator 1 as they contain and voted on their own version of the block in slot 3.

To find an explanation for the time, i.e., slot, for which the validators stop voting the test was run again with the option to print in a time interval rather than after a new observed maximum vote which gives us the state of the block stores shown in Figure 2. (Note: A block store of a validator can only store one block per slot). The behavior of validators 2 and 3 follows from the duplicate observation rule mentioned in Section 3.2. However, the validators might fork and keep creating blocks, but be unable to switch back their votes.

Regarding the question of why validators 2 and 3 do not join each other, a possible explanation is that neither of them can vote on their own block, and hence they only see blocks without votes and decide to create their own fork.

- **Tests 2–4 (**$(90\%, 5\%, 5\%)$ **and rotations):**

This case should show how the cluster behaves in a trivial case. In all the cases, we get the result that the validators decide on a single version of the block in slot 3, i.e., the version of the highest-staked validator. The time when the decision is made depends on the highest staked validator and its restart time. Only after the highest staked validator has been restarted, its vote on its version of the block in slot 3 is visible to others and results in a change of the block in slot 3, respectively.

A possible explanation for the behavior in test 2, is that validator 1's vote is immediately visible to the others since they are being restarted after validator 1. Hence, in these cases, the outcome is always the same: All validators adopt the block version of validator 1 and join its fork right away.

In test 3, it seems as if the first validator often already produced blocks for its version, as it has not noticed a duplicate yet. Although in all cases the validators decide on the block version of validator 2 and join its fork, we have the following different minor forks which are explained with the following reasoning:

1) (Tests 3.1, 3.5) Fork $[3] \longleftarrow [4, 5]$: Here the first validator forks itself while producing the slots in its schedule range from slot 4 to 8 and places a placeholder as block hash in for slot 4. This occurs since the validators decide on the block version of validator 2 while validator 1 was creating block 4 forcing them to discard its progress and continue on block 5 by directly appending it to the decided version of the block in slot 3.
2) (Tests 3.2, 3.3) Fork $[3] \longleftarrow [4, 8]$: If however validator 1 already produced its block in slot 4 based on its version of the block in slot 3, validator 2 may see this initial version of the block in slot 4 but obviously cannot verify it. In this case, validator 2 will replace all blocks from validator 1 with a placeholder hash, even after validator 1 replaced its block version in slot 3. Then, when it is validator 2's turn to produce blocks (slots 8–11), it will fork from slot 3 to append its blocks, with validators 1 and 3 joining at slot 13 (lockout timer on the vote of block 4 for 8 slots) and slot 8, respectively.
3) (Tests 3.2, 3.3, 3.4) Fork $[4] \longleftarrow [5, 6]$: Here the same happens as in 1) but validator 1 places a placeholder in slot 5 rather than slot 4.

Here it is also worth mentioning that the behavior of validator 1 of changing the block hash of a block after the decision of a unique block version of slot 3 is dangerous as it can introduce confusion for validators if multiple block hashes of the same slot get shared by the same validator.

In test 4, validator 3 is restarted last. It is in these cases where the behavior differs the most although in all runs the validators decide on the block version of validator 3. Here possible explanations for the forks are as follows:

1) (Tests 4.1, 4.2, 4.5) Fork $[3] \longleftarrow [4, 8]$: See 2) of test 3.
2) (Tests 4.1, 4.2, 4.4, 4.5) Fork $[4] \longleftarrow [5, 6]$: See 3) of test 3.
3) (Tests 4.3, 4.4) Fork $[3] \longleftarrow [4, 12]$: Here we have validators 1 and 2 joining each other while validator 3 creates a fork in the first slot of its schedule from slot 12 to 15. A possible explanation for the behavior of validator 3 is, that it could not verify the block hash of slot 4 and hence all following block hashes of the fork of validator 1 and 2, although validator 1 changed its block hashes later to match the chosen block version of slot 3 of validator 3.
4) (Run 4.3) Fork $[4] \longleftarrow [5, 9]$: Here validator 2 actually skipped the block in slot 8 and continued with block 9 directly.
5) (Run 4.3) Fork $[5] \longleftarrow [6, 7]$: This fork appears probably due to the decision on the block version occurring while validator 1 is producing the block in slot 6, forcing it to discard its progress and to continue on slot 7.

In run 4 we see the second validator stop voting at slot 11 while validators 1 and 3 join on the fork of the latter as shown in Figure 3. This could be because validator 2 reduces its lockout timers which got introduced by joining validator 1's fork and voting on the blocks in slots 3, 4, 6, 7, 8, 9, 10, and 11. Validator 2 possibly stops voting after its last leader slot to already reduce its lockout timers to join validator 3 earlier. It is however not entirely clear why validator 2 stops voting while validator 3 continues if they have the same state of the block store.

- **Tests 5–10 (**$(67\%, 16\%, 17\%)$**,** $(66\%, 17\%, 17\%)$ **and rotations):**

When comparing the outcomes of tests 5–7 with those of tests 8–10 we see no different behavior which leads to the conclusion that in this scenario a supermajority has no effect. We also have the same forks occurring as in the previous tests, these being:

1) (Tests 6.2, 6.4, 9.1) Fork $[3] \longleftarrow [4, 5]$: See 1) of test 3.
2) (Tests 6.2, 6.5, 9.3, 10.1) Fork $[3] \longleftarrow [4, 8]$: See 2) of test 3.
3) (Tests 6.2, 6.3, 6.5, 7.1, 7.5, 9.2-9.5, 10.1, 10.2) Fork $[4] \longleftarrow [5, 6]$: See 3) of test 3.
4) (Tests 7.1-7.5, 10.2-10.5) Fork $[3] \longleftarrow [4, 12]$: See 3) of test 4.
5) (Tests 7.2-7.4, 10.3, 10.4) Fork $[4] \longleftarrow [5, 9]$: See 4) of test 4.
6) (Tests 7.2-7.4, 10.3, 10.4) Fork $[5] \longleftarrow [6, 7]$: See 5) of test 5.

As in test 4.4, we reach the same block stores in tests 7.1 and 7.5 with validator 2 again abstaining from voting after slot 11. In tests 10.2 and 10.5, however, we have the same block store but now with two abstaining validators 1 and 2.

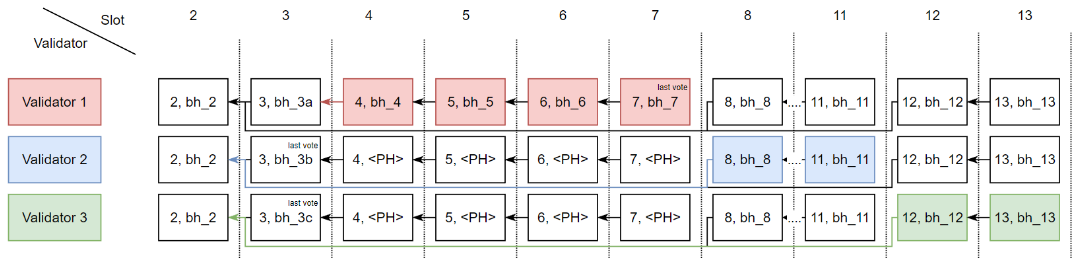

Figure 2: Visualization of the validators' state in the equal case. Colors indicate the block producer.

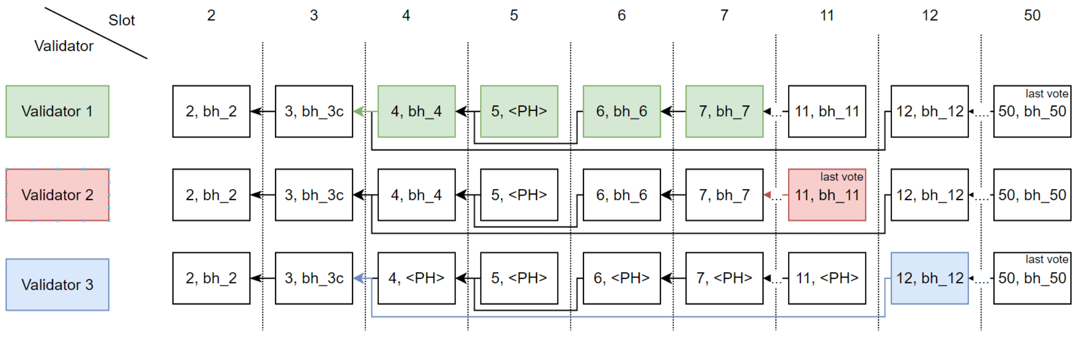

Figure 3: Visualization of test 4.4. Colors indicate the block producer.

A possible explanation for this behavior could be that now both validators voted up to slot 11 and hence introduced big lockout timers for both of them rather than one of them stop voting earlier. Here it is unclear how this behavior was influenced by this stake distribution change.

- **Tests 11–16 (**(39%, 30%, 31%), (38%, 31%, 31%) **and rotations):** Examining the outcomes of these stake distributions, we see that consensus was never reached. Possible explanations for the occurring forks are the following:
  1) (Tests 11.1, 13.4, 13.5) Fork [2] ⟵ [3, 8]: Here only validator 2 rolls back to slot 2 to create a fork while validator 1 continues producing and voting on its fork. Validator 2, however, is unable to roll back its vote to slot 2 due to an implementation bug, which is examined further in Section 5.
  2) (Tests 11.1, 13.4, 13.5) Fork [3] ⟵ [4, 12]: See 3) of test 4.
  3) (Tests 11.2, 11.4, 11.5, 14.3-14.5) Fork [2] ⟵ [3, 8, 12]: Validator 1 continues on its fork, unaware of the duplicate in its slot 3 while both validators 2 and 3 manage to roll back to slot 2 and create a fork.
  4) (Tests 11.3, 14.1) Fork [2] ⟵ [3, 12]: Similar to 3) but only validator 3 manages to roll back and fork at slot 3.
  5) (Tests 11.3, 14.1, 16.5) Fork [3] ⟵ [4, 8]: See 2) of test 3.
  6) (Tests 13.1, 13.2, 14.2, 15.3, 16.1) Fork [3] ⟵ [4, 8, 12]: None of the validators is aware of the duplicate hence everyone forks at slot 3 because none of them can verify other blocks received.

  In addition to forks, we also have completely different voting behavior. Possible explanations for these behaviors are the following:
  – (Tests 11.1, 13.4, 13.5) The highest staked validator continues to vote on its fork while the second validator abstains from voting right after its own slot 3 and the other remaining validator voting up to the last slot of its schedule:
    ∗ Validator 1 is unaware of the duplicate and simply continues.
    ∗ Validator 2 started a fork at slot 2 because it rolled back but is unable to switch its vote.
    ∗ Validator 3 forks at slot 3 because it is unable to validate any blocks of validator 1. Then after slot 15, it wants to switch forks but is unable due to not reaching the duplicate threshold.
  – (Tests 11.2, 11.4, 11.5, 14.3-14.5) The first, i.e., highest staked validator continues to vote on its fork while the other two validators stop voting right after slot 3:
    ∗ Validator 1 continues on its fork, unaware of its duplicate.
    ∗ Validators 2 and 3 manage to roll back to slot 2 and create a fork but are unable to vote.
  – (Tests 11.3, 14.1) The first, i.e., highest staked validator continues to vote on its fork while the second validator stops voting at slot 11 and the third right after its slot 3: Similar to 1) but with changed roles of validators 2 and 3.
  – (Tests 13.1, 13.2, 14.2, 15.3, 16.1) The highest staked validator continues to vote on its fork while the other validators vote up to the last slot of their own schedule:

  - ∗ Validator 1 continues on its fork, unaware of its duplicate.
  - ∗ Validators 2 and 3 fork at slot 3 but abstain after their first schedule, respectively, unable to switch to validator 1's fork due to missing duplicate threshold.

  In the remaining cases where all validators abstain, we can again inspect the block stores with the timed method instead of the method of the vote. We see that every validator that did not already fork at slot 3 will fork at slot 2. A possible explanation is that only the first validator, which is unaware of the duplicate at restart time, continues. The other validators roll back to slot 2, if possible, and get their own votes stuck due to the bug. It is also possible that validator 2 sees the duplicate late and already produced its blocks before abstaining.

- **Tests 17–22 (**$(52\%, 24\%, 24\%)$**,** $(53\%, 24\%, 23\%)$**, and rotations):**
  In these outcomes, we clearly see the impact of the duplicate threshold. For tests 17-19, consensus was achieved rarely, and the same forks can appear as mentioned before. For tests 20-22 nearly every run ended in a consensus with some minor forks appearing since the validators have simply the ability to vote on the highest staked fork and join it straight away. The forks appearing are explained as follows:
  1) (Tests 17.5) Fork $[2] \longleftarrow [3, 8, 12]$: See 3) of tests 11-16.
  2) (Tests 18.1) Fork $[2] \longleftarrow [3, 12]$: See 4) of tests 11-16.
  3) (Tests 18.1, 21.1, 21.2, 21.4, 22.2, 22.3) Fork $[3] \longleftarrow [4, 8]$: See 2) of test 3.
  4) (Tests 18.3, 18.5) Fork $[3] \longleftarrow [4, 8, 12]$: See 6) of tests 11-16.
  5) (Tests 19.3, 22.1, 22.4, 22.5) Fork $[3] \longleftarrow [4, 12]$: See 3) of test 4.
  6) (Tests 21.5) Fork $[3] \longleftarrow [4, 5]$: Here the same happens as in 1) of test 3 but one block earlier.

  In addition to forks, we also have different voting behavior. Possible explanations for these behaviors are the following:
  – (Test 17.5): Same as voting behavior explained for tests 14.3-14.5.
  – (Test 18.1): Same as voting behavior explained for test 14.1 but with highest staked validator 2 instead of 1.
  – (Tests 18.3, 18.5): Same as voting behavior explained for test 15.3
  – (Tests 19.3, 22.5) Validators 1 and 2 stop voting after the last block of validator 2's schedule while validator 3 continues on its fork: In both tests, the validators want to reduce their own lockout timers in order to join validator 1, which will only be possible in test 22 due to the missing duplicate threshold in test 21. It is however unclear why validators 1 and 2 see the fork of validator 3 in test 19 since it should not be possible to vote due to the missing duplicate threshold.

All the results were obtained with the same test implementation and were run five times. Still, by only changing the stake distribution, the outcomes vary greatly, sometimes even between runs with the same parameters. For all cases where no validator reaches the duplicate threshold, the outcome is unclear. Since no fork based on a duplicate is votable under this threshold, the nodes would rely on rolling back to the block before, which rarely works. All in all, it seems as if the mechanism is unreliable for stake distributions under the duplicate threshold, ending up in non-deterministic behavior.

#	Stake 1	Stake 2	Stake 3	Outcome
1	333,333	333,333	333,333	Fork (Stuck)
2	900,000	50,000	49,999	Resolved
3	49,999	900,000	50,000	Resolved
4	50,000	49,999	900,000	Mixed
5	670,000	160,000	169,999	Resolved
6	169,999	670,000	160,000	Resolved
7	160,000	169,999	670,000	Mixed
8	660,000	170,000	169,999	Resolved
9	169,999	660,000	170,000	Resolved
10	170,000	169,999	660,000	Mixed
11	390,000	300,000	309,999	Fork (2 & 3 Stuck)
12	309,999	390,000	300,000	Fork (Stuck)
13	300,000	309,999	390,000	Fork (Mixed)
14	380,000	310,000	309,999	Fork (2 & 3 Stuck)
15	309,999	380,000	310,000	Fork (Mixed)
16	310,000	309,999	380,000	Fork (Mixed)
17	520,000	239,999	240,000	Mixed
18	240,000	520,000	239,999	Fork (Mixed)
19	239,999	240,000	520,000	Mixed
20	530,000	239,999	230,000	Fork (Mixed)
21	230,000	530,000	239,999	Fork (Mixed)
22	239,999	230,999	530,000	Fork (Mixed)

Table 2: Experiments in the *Duplicate Slot* scenario. Note that total stake is 1 million units in all tests as the malicious dummy validator holds one unit.

Slots	0	1	2	3	4	5	6	7	⋯	20
Leader	D1	D1	D2	D3	1	2	3	1	⋯	3

Table 3: Leader schedule for the *Shifted Blocks* scenario, where D1, D2, D3, denote the three dummy validators.

4.2 Shifted Blocks

In this experiment, a similar case to the one in Section 4.1 was analyzed, but instead of creating three blocks for the same slot, three different forks were created, each containing one of the blocks in the slots between 1 and 3.

Note that this scenario could arise purely from bad synchronization, without any adversarial behavior by any validator.

At the beginning, three validators with epsilon stake produce an initial state. The first dummy validator is in charge of producing the first block, which every other validator agrees on. Then the dummy i produces the block for slot i, meaning that we have three forks with only one block starting at slot 0. These individual forks then get copied to the three validators whose behavior we want to observe. An example initial state is shown in Figure 4. The three

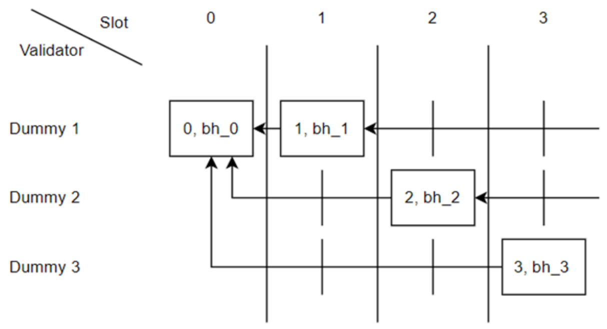

Figure 4: Initial state of the *Shifted Blocks* scenario. The three blocks in slots 1, 2, and 3 were all created by the correct leaders (dummies 1, 2, and 3 respectively). The three non-dummy validators start with the same blocks as the dummy validator of their respective number.

validators are then restarted, left to vote on their own fork, and run for 20 slots. This test was run multiple times with changing stakes of the three validators.

- **Test 1** $(33.\overline{3}\%, 33.\overline{3}\%, 33.\overline{3}\%)$**:**
  In contrast to test 1 in Section 4.1, the validators here always reach consensus on the fork of validator 3. Although a specific rule regarding the scenario could not be found, a possible explanation for the decision, is the age of the block which is the lowest for the one of validator 3. In these cases, validator 2 cannot finish its first block in slot 5 as it simply puts a placeholder in there, possibly seeing the newest vote of validator 3 at this moment.
- **Tests 2–10 (**$(90\%, 5\%, 5\%)$**,** $(67\%, 16\%, 17\%)$**,** $(66\%, 17\%, 17\%)$ **and rotations):**
  In all these cases and their rotations, we always achieve consensus in the same way and always as expected, i.e., agreeing on the fork of the highest staked validator.
- **Tests 11–16 (**$(39\%, 30\%, 31\%)$**,** $(38\%, 31\%, 31\%)$ **and rotations):**
  Only considering the problematic cases, i.e., where the second validator has the highest stake in tests 12 and 15, we observe that although consensus is being achieved, the validators decide on the fork of validator 1. Here validator 2 always produces a placeholder in slot 5 before directly joining validator 1's fork. Since this outcome has been observed in every run without the slightest changes, additional adaptions were created. These adaptions are explained below.
- **Tests 17 and 18 (**$(19\%, 62\%, 19\%)$**,** $(20\%, 61\%, 19\%)$**):**
  Here the difference is clearly visible because in test 17 all the runs end with all validators joining validator 2's fork, while in test 18 they also achieve consensus but on the fork of validator 1. But as mentioned in Section 3.2, such a threshold of 61% could not be found in the implementation.

4.3 Adaptions

We tried two adaptions of the most interesting experiments in the *Shifted Blocks* scenario. The first adaption sees validator 2 vote on its initial block, i.e., in slot 2 before disabling its ability to vote. This needs to be done rather than terminating as validator 2 can still produce blocks and requires propagating its information to the other validators. But without its vote, validator 2 will not interfere with the outcome of the two other validators. This case should answer the question if the behavior of always joining the first validator is connected directly to validator 2 or not.

- **Test 11–13 (**$(39\%, 30\%, 31\%)$ **and rotations):**
  For the runs where the second validator has the majority of the stake, i.e., test 12, we observe two different outcomes, both resulting in validator 3 abstaining from voting. In the first outcome observed in run 3, validator 2 forks slot 3 twice, the second time being on slot 5 where it also continues to produce blocks. Validator 1, however, continues voting on its own fork with voting for each of its appended blocks. Validator 3 produces and appends its blocks on its fork but does not vote for them. It seems as though validator 2 is massively behind with block production that neither validator 1 nor validator 2 could vote on their own fork, respectively. Hence, validators 1 and 2 continue on their own fork, with validator 3 unable to join as the required switch threshold is not reached by the stake of validator 1. In the second outcome observed in all other runs, validator 2 directly produces blocks for the fork of validator 1 while validator 3 abstains from voting at slot 3 but still produces blocks for its fork. A possible explanation for the behavior of validator 3 could again be connected with the missing switch threshold of the fork of validator 1. But the behavior of validator 2 is not explainable to us.
- **Tests 14–16 (**$(38\%, 31\%, 31\%)$ **and rotations):**
  In these cases, the behavior even changed for the case where the first validator has the most stake, i.e., test 14, as its outcome is identical to run 4 where the second validator has the most stake as explained above.
- **Tests 17 and 18 (**$(19\%, 62\%, 19\%)$ **and** $(20\%, 61\%, 19\%)$**):**
  Although the first run of test 18 results in consensus on the fork of validator 2, we otherwise see a clear distinction. For test 17, we always see the validators joining validator 2's fork like in the unmodified scenario. In all but the first runs of test 18, we see validator 3 abstaining from voting after slot 3 while still producing blocks for its fork, while validators 1 and 2 stay on the fork of validator 1. A possible explanation for the behavior of validator 3 unable to switch forks, is the missing switch fork threshold of 38% and hence validator 3 waits for the first fork to gain additional stake. Compared to the unmodified scenario, the third validator also only switched forks right after validator 2 voted for it. In the first run, validator 2 continued to produce blocks on its fork allowing validator 1 to join on slot 7. The reason why validator 2 does not join validator 1 as in the other cases is not entirely clear to us. A possible explanation is that in this single run validator 2 saw the vote of validator 1 on slot 4 too late.

#	Stake 1	Stake 2	Stake 3	Outcome
1	333,332	333,332	333,332	Resolved
2	899,999	49,999	49,999	Resolved
3	49,999	899,999	49,999	Resolved
4	49,999	49,999	899,999	Resolved
5	670,000	160,998	168,999	Resolved
6	168,999	670,000	160,998	Resolved
7	160,998	168,999	670,000	Resolved
8	660,000	169,998	169,999	Resolved
9	169,999	660,000	169,998	Resolved
10	169,998	169,999	660,000	Resolved
11	390,000	299,998	309,999	Resolved (Unexplained)
12	309,999	390,000	299,998	Resolved (Unexplained)
13	299,998	309,999	390,000	Resolved (Unexplained)
14	380,000	309,998	309,999	Resolved (Unexplained)
15	309,999	380,000	309,998	Resolved (Unexplained)
16	309,998	309,999	380,000	Resolved (Unexplained)
17	189,999	620,000	189,998	Resolved (Unexplained)
18	199,999	610,000	189,998	Resolved (Unexplained)

Table 4: Experiments in the *Shifted Blocks* scenario. Note that total stake is 1 million units in all tests as there are three malicious dummy validator with one unit stake each.

From these results, we see that without validator 2 voting on additional blocks, validator 3 can get stuck as it cannot produce a switch proof due to lack of stake on validator 1's fork. But in the 62% case, the vote of validator 2 on its slot was enough for both other validators to join the fork of validator 2, even if validator 2 stopped voting.

The second adaptation is based on the first one but restores the voting behavior of validator 2 after ten slots. This should also explain whether the votes of validator 2 have an impact at all on a validator possibly waiting for enough stake to gather on a fork.

- **Tests 11–13 (**$(39\%, 30\%, 31\%)$ **and rotations):** Compared to test 12 in the first adaption, we see that the third validator joins validator 1 after it observed the vote of validator 2 on the fork of validator 1. This is probably due to the fork of validator 1 reaching the switch threshold as soon as validator 2 votes on it. We also have one outlier in the third run of test 13 as validator 2 never leaves its own fork, although validator 1 joined validator 3's fork already earlier.
- **Tests 14–16 (**$(38\%, 31\%, 31\%)$ **and rotations):** Here we observe the same behavior as above but now with run 1 actually agreeing on the fork of the highest staked validator 2. This outcome is the expected behavior of the scenario, but it is unclear why it was only achieved once.
- **Tests 17 and 18 (**$(19\%, 62\%, 19\%)$ **and** $(20\%, 61\%, 19\%)$**):** Here we have the same clear distinction as in the adaption before. Furthermore, in test 18 we now see validator 3 returning to vote on validator 1's fork as soon as validator 2's vote functionality is restored after slot 10, which underlines that validator 3 is waiting for more stake to gather on the fork of validator 1.

The question of the origin of the 61% threshold still remains. Further it seems as if there are additional rules making validators abstain from voting to await more stake to gather on one of the forks. The answer to this behavior and the threshold could not be found at the time of this work and is hence left as potential future work.

5 REAL-WORLD EXAMPLE

On the 30th of September 2022, the Solana mainnet experienced an outage. According to the post-mortem,[11] the outage lasted around eight hours and required centralized manual intervention to fix. They also confirmed that on slots 153,139,220 and 153,139,221 duplicates were observed, due to a configuration error on the single validator, which was the leader responsible for these slots. At first, some validators voted on one duplicate while not seeing the other. Other validators who observed both versions abstained from voting, rolled back to the block before, and started a new fork. Unfortunately, the majority voted on the same version of the duplicate forcing the validators on the newly created fork to switch. However, due to an implementation error in the *Heaviest Subtree Fork Choice* module, the validators were unable to do so.

This outage is related to our *Duplicate Slot* scenario in Section 4.1. We reevaluated the scenario on a newer commit[12] including the proposed fix mentioned on the Solana news page.[13] The suggested fix, however, did not solve the strange behavior observed in Section 4.1. In our test with logging enabled we observe the logs mentioned by the involved validator.[14]

6 RELATED WORK

Many works exist performing in-depth theoretical analysis of the security and liveness properties of popular blockchains, most notably regarding Bitcoin [3, 9, 10] and Ethereum [5, 13, 18]. Also, various comparative studies of consensus protocols employed by blockchain systems exist [1, 4, 6, 17].

However, as previously mentioned, there has been almost no academic research on the Solana blockchain so far. Specifically, to the best of our knowledge there exists not a single work critically examining the consistency and correctness of consensus rules in play in Solana. We are not aware of any work that provides formal proofs of correctness, security, or liveness for the protocol; this includes the Solana whitepaper itself. One prior work analyzing the blockchain protocol, focuses solely on its scalability [12].

The most important related works are the ones that build the foundation of understanding the core principles of the Solana implementation such as the "Proof-of-History" whitepaper [19]. This whitepaper is somewhat outdated, but explains the essentials of the blockchain protocol and system architecture.

One critical analysis of Solana's inner workings is the write-up by Shoup [14], which focuses on the Proof-of-History mechanism

[11]https://solana.com/de/news/09-30-22-solana-mainnet-beta-outage-report
[12]https://github.com/solana-labs/solana/commit/4cbf59a5ddd31e4cbcd545e128b9e459cf56b036
[13]https://github.com/solana-labs/solana/pull/28172
[14]https://youtu.be/7tGYT8j7AbE?t=1002

in detail. The key findings include that Solana's Proof-of-History does not constitute a VDF [2] as claimed by the original authors, since it lacks a "[...] verification algorithm [that] takes much less computation than that of the prover, without relying on parallelization" [14]. Further, it calls additional claims of the whitepaper into question and calls out the inherent waste of energy of this approach, as it undermines one of the main advantages of consensus protocols based on Proof-of-Stake over those using Proof-of-Work.

7 CONCLUSION

In the results of the experiments in Sections 4.1 and 4.2, we came across some to us not explainable behavior of the validators. Especially the *Duplicate Slot* scenario for stake distributions $\leq 52\%$ in Section 4.1 has raised more questions than it has answered.

In many places it was hard for us to bring differences between the whitepaper, documentation, and code together into a coherent picture. Luckily, the implementation features many test cases, especially edge case related. The actual code thus proved invaluable for reverse engineering the protocols rules.

From a standpoint of a reliable decentralized network, the repeated outages requiring centralized manual intervention are concerning. Even though Solana is officially in a Beta state, it has a market capitalization of over $US 20 billion and is therefore ranked 7th (as of November 2023) out of all cryptocurrencies.[15]

Solana takes a non-standard approach to achieving state replication in view of possible (adversarial) failures, making strong assumptions about the failure cases that can arise in practice, that seem to go far beyond the normal theoretical bounds for these problems. Based on the tests in Figure 2, we suggest that duplicate handling, especially with the largest fork at $\leq$ 52% stake is not fully handled by the consensus protocol. In view of the November 2022 outage, the likelihood of this problem arising in practice even without malicious intent should be reevaluated.

In the absence of such a concrete implementation, it is accepted that centralized manual intervention will be required in the event of a malfunction. Underlying this assumption is the fact that the presented duplicate handling was only first addressed in March 2021, around one year after the mainnet ledger started.[16]

Future Work

The same setup as ours, using the *LocalCluster* module, can be employed to explore a variety of scenarios. Hence, future works could analyze other edge cases and try to more fully understand the protocol behaviors. Continuing on the experiments conducted in this work, an extension of the *Shifted Blocks* experiment to better understand and also explain some outcomes would be useful, especially regarding the found threshold of 61% and the abstaining behavior of validator 3 in the adaptations.

It would also be interesting, to better quantify the risks of denial-of-service attacks based on the protocol behaviors laid out in this work. To this end, one should probably also take a closer look at Turbine, Solana's block propagation mechanism.

We hope that our findings will be helpful in informing design decisions for the Solana project as well as other blockchain systems going forward.

Apart from the direct results of this work regarding the correctness and liveness of the consensus protocol in the Solana blockchain, we hope that our findings regarding the consensus rules may serve as an additional resource for future researchers trying to understand the protocol. Specifically, they may lead to a more complete understanding than it seems possible from the whitepaper alone.

Acknowledgements

We would like to thank Thomas Locher for his continued guidance and valuable suggestions during the project, as well as his feedback, which helped improve the presentation of this work.

REFERENCES

[1] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 183–198.

[2] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Annual international cryptology conference*. Springer, 757–788.

[3] Joseph Bonneau. 2016. Why buy when you can rent? Bribery attacks on bitcoin-style consensus. In *International Conference on Financial Cryptography and Data Security*. Springer, 19–26.

[4] Christian Cachin and Marko Vukolić. 2017. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[5] Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. 2022. Goldfish: No More Attacks on Proof-of-Stake Ethereum. *Cryptology ePrint Archive* (2022).

[6] Md Sadek Ferdous, Mohammad Jabed Morshed Chowdhury, and Mohammad A Hoque. 2021. A survey of consensus algorithms in public blockchain systems for crypto-currencies. *Journal of Network and Computer Applications* 182 (2021), 103035.

[7] Solana Foundation. 2023. Optimistic Confirmation and Slashing. https://docs.solana.com/de/proposals/optimistic-confirmation-and-slashing. Official Solana Documentation.

[8] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*. 51–68.

[9] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th usenix security symposium (usenix security 16)*. 279–296.

[10] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. 2020. BDoS: Blockchain denial-of-service. In *Proceedings of the 2020 ACM SIGSAC conference on Computer and Communications Security*. 601–619.

[11] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* (2008).

[12] Giuseppe Antonio Pierro and Roberto Tonelli. 2022. Can solana be the solution to the blockchain scalability problem?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1219–1226.

[13] Caspar Schwarz-Schilling, Joachim Neu, Barnabé Monnot, Aditya Asgaonkar, Ertem Nusret Tas, and David Tse. 2022. Three attacks on proof-of-stake ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 560–576.

[14] Victor Shoup. 2022. Proof of history: What is it good for? (May 2022).

[15] Solana Foundation. 2022. 09-30-22 Solana Mainnet Beta Outage Report. https://solana.com/news/09-30-22-solana-mainnet-beta-outage-report. Official Solana Blog.

[16] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[17] Yang Xiao, Ning Zhang, Wenjing Lou, and Y Thomas Hou. 2020. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials* 22, 2 (2020), 1432–1465.

[18] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. 2023. Speculative Denial-of-Service Attacks in Ethereum. *Cryptology ePrint Archive* (2023).

[19] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper* (October 2018).

[15]https://coinmarketcap.com
[16]https://github.com/solana-labs/solana/pull/16127