University of Stavanger

DANIEL S. HAVSTAD
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Evaluation and Revision of Swarm's Redistribution Smart Contract

Master's Thesis - Computer Science - June 2024

```go
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
	if len(opts) < 1 || len(opts) > 2 {
		return nil, fmt.Errorf("wrong number of options: %d", len(opts))
	}
	c = &Configuration{}
	for _, opt := range opts {
		switch v := opt.(type) {
		case gorums.NodeListOption:
			c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
			if err != nil {
				return nil, err
			}
		case QuorumSpec:
			// Must be last since v may match QuorumSpec if it is interface{}
			c.qspec = v
		default:
			return nil, fmt.Errorf("unknown option type: %v", v)
		}
	}
	// return an error if the QuorumSpec interface is not empty and no impl
	var test interface{} = struct{}{}
	if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
		return nil, fmt.Errorf("missing required QuorumSpec")
	}
	return c, nil
}
```

I, **Daniel S. Havstad**, declare that this thesis titled, “Evaluation and Revision of Swarm’s Redistribution Smart Contract ” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master’s degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

“Programming is a nice break from thinking.”

– Leslie Lamport

Abstract

Decentralized peer-to-peer (P2P) storage networks using blockchain technology such as Swarm emerge as a viable alternative to central cloud storage. Providing a solution to having a single point of failure, added security by not having to trust third party cloud providers, and resistance against possible censorship. However, ensuring the longevity, reliability, and fairness of such networks presents formidable challenges in the face of node churn and free riding. Each network participant needs to be compensated accordingly for both their storage capacity needed to persist user files, and the bandwidth needed for clients to upload and retrieve their files. Swarm in particular aims to be a zero cost of entry, self regulating, and sustainable storage network, boasting that storage incentives is the missing piece for blockchain. The Swarm storage incentives are handled by a redistribution game that is run by a set of Ethereum compatible smart contracts. Each round of this game decides on a network participant to receive the reward for storing files through the redistribution smart contract.

In this thesis we are evaluating the storage incentives in Swarm by analysing the redistribution smart contract. Our analysis shows that the current truth selection and freezing mechanisms in Swarm, lead to a viable free riding strategy for malicious storage nodes. We propose two alternative solutions to mitigate the problem: the alpha solution, and the bank solution. Both solutions have the attribute, that the reward in a redistribution round might not be handed out to storage node network participants. In the case where multiple different proofs of storage are submitted in a round, thus providing incentive for every network node to work together in submitting the same value. We refer to the case of when there is no winning storage node, as a win for the bank entity. The bank solution is more simple, and it is easier to implement such that it minimizes the gas cost of the contract. Whereas the alpha solution is overall more fair, but trickier to implement, and with more gas overhead.

Further key milestones in this thesis are that. We show how both solutions can be implemented in the Solidity programming language. That we evaluate the implemented solutions by using real data from previous redistribution rounds. In this evaluation we showed that the implemented smart contracts could be run in a realistic albeit development setting with ganache blockchain. And we were able to confirm that we were effective in mitigating the strategy of sending in arbitrary proof of storage, whilst comparing the two different solutions. When deciding what to do when the bank entity wins, we take care to analyse what happens if we burn or carry the

reward over into the next round. Particularly interesting is the latter case (which we decided to implement), where new possible storage incentive exploits could occur. These are exploits that rely on a node operator being present in multiple Swarm neighbourhoods, to try and increase their chance to win the carried over reward. We looked into ones where said node operator had i) the same stake in each neighbourhood, and ii) dividing the neighbourhoods the operator is in by two, having a separate stake in each. The majority of exploits we looked at needed an unfeasible amount of investment to pull off for bank, and alpha. However we did discover that the bank solution is vulnerable to ii).

Acknowledgements

I would like to thank my supervisor Leander Nikolaus Jehl for their enthusiasm, advice and overall guidance while working on this thesis.

I thank my parents, and my brother for their unwavering support, which has provided me with the strength necessary to complete this journey.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

The most prominent solutions for distributed storage today are cloud storage platforms such as Google Drive [1], or Microsoft OneDrive [2]. These services provide users access to their files anywhere, anytime, and do so while potentially reducing the time and money someone would need to spend to persistently store their files locally. The catch is, however that you need to trust the service provider with your files, and should they fail or go bankrupt then your data might not be recoverable. Recently Google has had to deal with backlash over lost files on their Google Drive [3]. A solution against this single point of failure is instead of using a client-server model, to use a peer to peer (P2P) network in order to have decentralized storage.

Swarm is a decentralized storage network where participating nodes called bees organised into neighbourhoods, store user data by splitting an uploaded file into 4KB chunks that are stored and replicated on the network [4]. Other, similar actors to Swarm in decentralized storage are Filecoin[5], Storj[6], and Sia[7]. These networks are utilizing the properties of blockchain, to provide a ledger that keeps track of nodes storing files, and/or keep track of storage transactions between client and storage node. Each network conduct transactions using their own token or cryptocurrency, BZZ being Swarm's token.

Currently it is difficult for most to say they own the full stack of their application, and often times it is the hosting of the application that is missing [8]. This is one thing decentralized storage can be used for, and right now it is possible to host a website with storage on Swarm [9]. In particular this can solve possible censorship by third parties or governments. Citizens in China, Russia, South Korea, Pakistan, Egypt and more, are experiencing a censored Wikipedia [10]. Since Wikipedia is a central and public organisation, it is possible to put pressure on them to take down or alter their content. However, by hosting Wikipedia on Swarm then you would have to pressure the whole network, and it allows for the possibility of forking Wikipedia [8][11].

Swarm can look to the success and failings of the P2P file sharing service BitTorrent. Notably while the tit for tat file-sharing in BitTorrent is cost-effective, it lacks incentives for nodes to keep sharing files after they have completed their download [12]. Which makes it hard for the system to persist files long term, especially those with lower popularity. Thus suggesting the need for financial incentives instead to allow for both sustainability and responsibility in file storage.

Additionally if decentralized storage is to become a proper alternative to cloud, then they need to find good ways to handle a central problem. Node churn, that is storer nodes may arbitrarily choose to leave the network. And when that happens data is at best unavailable, or lost forever [13]. As it stands the way to deal with node churn is to store data redundantly on many nodes, the caveat being that such storage networks are then reliant on having a lot of participating nodes. Furthermore if there is a crash in the token economy, or a sudden surge of popularity from a competing network, then it may cause a mass exodus of nodes. Which ultimately leaves the storage situation volatile. It is then all the more vital to have robust incentives, in order to keep nodes participating.

Swarm seeks to become a sustainable platform by (unlike BitTorrent) creating financial incentives for nodes taking part in the network. And by doing so avoiding a great deal of node churn in the long run. There are two kinds of financial incentives in Swarm. Firstly there's the bandwidth incentives that compensate nodes for their internet usage. Secondly there's the storage incentives which reward nodes for correctly storing the chunks. Swarm is adamant that storage incentives are the missing piece in order to make blockchain technologies viable [8].

1.2 Objectives

Exploring the effectiveness of, and improving upon Swarm's storage incentives will be the main focus of this thesis. Currently, Swarm utilizes Ethereum smart contracts, including the PostageStamp, PriceOracle, Staking, and Redistribution contracts, to manage storage incentives by handling network participation and rewarding BZZ tokens. Specifically, the redistribution smart contract decides how the rewards are allocated among network participants [4]. The redistribution of rewards play out as a coordination game using a commit-reveal scheme. Where all storage nodes in a neighbourhood commit a proof of storage for all their stored chunks, to later reveal. Ideally all nodes should reveal the same proof of storage, but there have been numerous cases where there are many different proofs revealed in a neighbourhood. It is believed that some of these different proofs are intentionally fabricated. The objective is to analyse the existing redistribution smart contract by the Swarm team, and investigate what incentive there could be to fabricate proofs of storage. Subsequently, if any such incentives are discovered, the aim is to propose and evaluate any potential solutions to mitigate the problem. In summary, the goal is to conduct a comprehensive analysis of the redistribution smart contract and propose how it can be further insulated from malicious actors.

1.3 Approach and Contributions

We analyse the existing redistribution contract mathematically. In this analysis it was found that the current freezing mechanisms of Swarm give possible incentive to send in fabricated proofs of storage. During truth selection the false proof of storage has a chance to be selected proportional to node stake as all other revealed values. And if selected all other nodes will be freezed, unable to participate in upcoming rounds. The increased likelihood for the malicious node to be freezed is offset with the ability to freeze the others and be the only node eligible for rewards in that neighbourhood in upcoming rounds. As such we show that the honest strategy is only a weakly dominating nash equillibrium, as this malicious strategy is equal to it. Furthermore if we consider that the malicious strategy does not necessarily need to store chunks, its utility increases.

We propose two alternative redistribution contracts, namely the ”Bank” and ”Alpha” contracts. Each alternative contract was designed to address the freezing mechanism problem identified during the mathematical analysis of the existing contract. The problem is solved by giving a bank actor the chance to win the reward in neighbourhoods where different proof of storage have been revealed. The ”Bank” contract prioritizes efficiency and scalability, while the ”Alpha” contract emphasizes enhanced security and fairness in reward distribution.

We extensively discuss what to do with the reward given to the bank (burn or keep), and mathematically analyse the expected reward when the reward won by the bank is carried over into the pot for subsequent rounds.

The proposed alternative contracts were implemented in Solidity, the programming language for Ethereum smart contracts. In particular finding solutions to avoid floating point operations, as these are not supported by Solidity, such as using the Babylonian method for roots, and multiplication instead of division. Ultimately trying to get as high performance, i.e lower gas costs as possible.

We test the implementation using truffle firstly by a cooked up example. And secondly by using real data of previous redistribution rounds from the existing contract, where the data was obtained from the Swarmscan API [14]. Our goal being to test that the implementation is flexible enough to handle any potential redistribution rounds. And to evaluate differences in the two alternative contracts ”Bank”, and ”Alpha”.

1.4 Outline

This thesis first begins by presenting relevant background information, and theory related to the Swarm decentralized storage network. Starting from more generalised information then narrowing it down to the focal point of the thesis, the Swarm storage incentives. We then analyse the current Redistribution contract, and discuss potential improvements. After which we implement the new Redistribution contract, and test it. Finally we discuss the results of our testing, and conclude the thesis. The content of each chapter is as follows:

- Chapter 2 Related Work, covers what research has previously been done regarding Swarm, and storage incentives.
- Chapter 3 Swarm, presents information about how the Swarm network is built in a bottom up manner, ending in an overview of the storage incentives.
- Chapter 4 Approach, starts by covering the existing redistribution contract in detail in section 4.2. We then analyse the contract mathematically using a Markov chain to prove what we believe to be a vulnerability. After which we discuss what our approach will be to fix the vulnerability.
- In chapter 5 Implementation, we show how the approach in chapter 4, is implemented in Solidity.
- In chapter 6 Experimental Evaluation, we present the test setup for testing our implementation. And the results of testing.
- Chapter 7 Discussion, Looks at the research findings found during the thesis.
- Chapter 8 Conclusions, provides a short conclusion of the thesis as a whole.

Chapter 2

Related Work

In this chapter we look at previous work done related to Swarm and storage incentives in decentralised storage networks. Furthermore we review the state of current research on decentralised storage networks by investigating the most recent surveys on the area. In addition to Swarm, these surveys consider the Storj, Filecoin, and Sia networks. Finally we do our own review of the aforementioned alternatives to Swarm, by looking through their respective documentation. We do this in order to get a grasp of other ways to do storage incentives than those present in Swarm.

We will now begin exploring research related to the Swarm decentralised storage network. The work done by Lakhani et al. studied the fairness of the reward distribution for Swarm's bandwidth incentives. They built a simulation tool, and simulated a Swarm network of 1000 nodes. The Lorenz curves with their corresponding gini coefficient was used to represent the fairness of the incentives
[15].

In the previous year, Kristian H. Tjessem in his master thesis developed a simulator coded in Go that simulates the Swarm network and storage incentive. By using the gini coefficient as a measure, Tjessem studied how fair reward distribution was given different network configurations and strategies [16].

In the paper by [13], they studied the data retention of Swarm, and found that an uploaded 5MB file could be irretrievable within less than a month. This due to key chunks going off the network. To solve this they proposed a Storage Upkeep Protocol (SUP), the protocol comprises of three agents, challenger, prover, and verifier. The challenger would ask the prover to prove that they stored a chunk. The proof sent by the prover would then be verified by the verifier. During file reupload, it can then be figured out exactly which chunks need to be reinstated, which saves bandwidth.

While the following paper is general purpose in nature, and not about Swarm specifically. it is relevant as a possible approach to storage incentives. Tas and Boneh propose a protocol for what they call Data Availability Committees (DAC). The definition of which is an off-chain system, for persisting and accessing data. This is done to reduce the cost of storing data on-chain, but the DAC can no longer rely on the security of the blockchain. Decentralized storage networks, Swarm included, have a DAC consisting of storage nodes in the network. The issues of DAC is trusting member nodes to store data. And even when correctly storing the data, trusting those same nodes will not deny legitimate requests for the data. They propose a solution to the problem of data availability that uses financial incentives. Slashing nodes that withhold data. If a client does not receive data within a timeout off-chain, then the next step of the protocol is to send the data query on-chain. The nodes who do not respond to the on-chain challenge, have their blockchain stakes slashed [17].

We found two papers mentioning Swarm, that provide recent insight into the area of decentralised storage as a whole. The most recent being a Systematization of Knowledge (SoK) conducted this year (2024). While the other paper is a survey from 2020.

The SoK by Chuanlei et al (2024), covers the topic of decentralized storage networks. They describe them as a paradigm shift for data storage. The SoK focuses on proof of storage, and storage incentives in the networks. They first describe the architecture of a decentralised storage network in general, and then compare the various storage networks: Sia, Storj, Filecoin, FileDAG, and Swarm. In particular they look into how the different networks do proof of storage, and which consensus algorithms are used for transactions. They have labeled Swarm as proof of work, but Swarm is using Ethereum based networks with proof of stake. Furthermore they are not covering the storage incentives in Swarm, although they do mention the Swarm bandwidth incentives. We take this as an indication that description of the storage incentives are not currently readily available [18].

In 2020 Zahed Benisi et al conducted a survey of decentralised storage networks mentioning Storj, Filecoin, Sia, and Swarm. Although a good survey at the time due to the rapidly evolving nature of this scene. Information from 4 years ago regarding the storage networks is already likely to be quite outdated. In regards to the main conclusions on disadvantage and benefits of decentralized storage and centralized cloud storage providers however, then the insight is still useful. A main issue blockchain networks face is scaleability. They highlight that it is possible to build reputation systems for storage nodes in these blockchain systems, as in the choice to only allow reputable nodes to store your data. Although there are various issues for how to do reputation in the most correct manner. The survey is focusing mostly on Storj [19].

The following is an evaluation of the other decentralized storage networks out there besides Swarm. We review the Filecoin, Sia, and Storj networks by looking through their official documentation, and papers.

Filecoin is built on the InterPlanetary File System protocol IPFS. The main difference from Swarm is that instead of levying more passive storage incentives FIlecoin uses the idea of a storage market. Additionally the market has three agents, the client, the storage providers (storage nodes), and retrieval providers. Retrieval providers provide bandwidth for a client to retrieve data from the storage node. Clients make deals with storage and retrieval providers, buying their solicited services for FIL. In a deal among other terms price, size of storage, and duration of storage are negotiated. The deal negotiation itself happens off-chain, and is published to the Filecoin blockchain upon agreement [5].

Sia also follows the marketplace principle unlike Swarm, where renters negotiate contracts on the blockchain with storers for storage. It also differs from Swarm by using a proof of work blockchain. Storage nodes set the prices and the idea is that free market rules of supply and demand will manage the price. Where Sia differs from Filecoin is that the Sia renter software renterd, will automatically form contracts for the client, based on their parameters [7].

In Storj's whitepaper they state that the system cannot be subject to restrictions by waiting for a blockchain to agree and process transactions, and propose a game theory approach. Storj differs from Swarm by using a reputation based system with satelittes. Their core design philosophy is that clients should avoid untrusted storage providers. Storj network comprises of storage nodes, and satellites. Satellites act as a network moderator for storage nodes, storage nodes get their payment from satellite nodes (they can work for multiple satellites). And satellites also manage the reputation of nodes, they will send random audits to storage nodes to check if they are faithfully storing data. In the case of failure a storage node will risk expulsion, and get no more payments from the satellite [6].

In most cases a storage node is assumed to be a reliable service providing node, this comes with the side effect that any new node must prove themselves. And as a result receive limited business opportunities. For them there is a vetting process, where the satellite will choose some unvetted nodes to store file data that does not impact the ability to retrieve that data. This in addition to a proof of work system, a filtering system, and a preference system is part of storej's reputation handling [6].

Chapter 3

Swarm

In this chapter, we will delve into the design principles underlying Swarm, offering essential context for understanding its storage incentive mechanisms. Following an overview, we will describe the Swarm Network layer, and how the routing between storage nodes work. Furthermore we will explain how files are organised into chunks for storage. And lastly the overall design of the Swarm storage incentives.

Swarm is a p2p network of storage nodes, that aims to let people share their left-over storage capacity for financial gain [8]. Users can pay to store files on the network. Swarm is designed as a layered architecture, starting from the bottom the layers are as follows:

- 1. underlay p2p network
- 2. overlay network and storage
- 3. data access API
- 4. application layer

1): Underlay p2p network refers to the protocol that allows direct network communication over IP to storage nodes. 2): Is the Swarm storage network itself, nodes, content addressing, file storage, and blockchain storage incentives. 3): Handles access to stored files.

And 4) being for people building applications on top of Swarm [20]. Of these four layers the developers of Swarm are mostly concerned with layers 2), and 3), calling those the Swarm core. This thesis concerns itself with 2) overlay network and storage.

3.1 Network

3.1.1 Underlay p2p network

While the specific underlay p2p network 1) used is up to each node to decide, there are some requirements the transport protocol needs to fill decided by the Swarm team.

Sent messages need to have guaranteed delivery, and messages need to be encrypted and authenticated. Swarm base their upper layer implementation on the libp2p library that satisfies all the requirements [21] [22]. These requirements are described in further detail in the book of swarm [20]. However, the main takeaway is that nodes in the underlay network are identified by their **underlay address**. It is this underlay address peers use to connect to each other.

3.1.2 Overlay network

Moving to layer 2), in addition to the underlay address, nodes are also identified by a unique 256 bit **overlay address**. The address is created by using the Ethereum account public key and hashing it together with the bzz network ID for Swarm. This network ID is different for testnet, and mainnet Swarm, giving users different overlay addresses on each. The overlay address is used to determine what peer a node will connect to, and the path messages are forwarded in the underlying p2p network [20].

The overlay network uses a Kademlia topology, which can ensure that there is a network path between any two nodes in $O(log(n))$ hops. Allowing the network to send messages using underlay network peer connections. Kademlia is a distributed hash table (DHT) [23]. The **Kademlia table** in Swarm indexes peers using the **Proximity Order (PO)** of the peer address relative to the nodes own overlay address [20]. PO is a measure of how related two addresses are, $PO(x, y)$ counts the matching bits starting from the most significant bit between the addresses until a bit no longer matches. In a 256 bit address space, the minimum distance then becomes 256, and the maximum 0. The table is organised into **PO bins**, peers with PO 255 are kept in the same bin and continuing on.

Now if a node has at least one peer in every bin until PO bin d_x, then it has a saturated Kademlia table. d_x is called the **neighbourhood depth**, and nodes within that depth are the **nearest neighbours** of a node. For routing purposes a saturated Kademlia table should always be satisfied. Because any peer could at any point go offline then it is not enough to only have a single peer in each bin [20].

Routing

Swarm uses what they call forwarding Kademlia. In forwarding Kademlia for a node to reach their destination, they will find the closest peer to that destination in their Kademlia table. And then that node will do the same forwarding the message at least one PO closer each time, until the destination is reached. The return trip can then follow the same path [20].

3.2 Chunk storage

3.2.1 Addressing

When a file is uploaded to the network, the file is split into 4KB chunks. Each chunk has an address that coincides within the same address space as the nodes themselves. The idea is that each chunk is stored on the nodes that are closest to the address. One reason for splitting the file is to help achieve storage load balancing between nodes. Chunks are spread uniformly in the address space, and as such it makes load balancing simpler when their size is fixed at 4KB. Another reason is that with the smaller size it allows concurrent retrieval even for small files in a manner similar to BitTorrent.

There are two different kinds of chunks in swarm. The first is a content addressed chunk, whose address is calculated by hashing an 8-byte span with the root of a Binary Merkle Tree (BMT) of the data. It is known as a BMT chunk. The data in the chunk is further segmented into 32-byte segments, the 8-byte span is the number that says how many of these segments hold actual data, the rest are padded to fill the 4KB chunk. The segments are put into the BMT to create a verifiable inclusion proof for the segments, that can be proven in $O(log(n))$, where n is number of segments.

The second type of chunk is a single owner chunk, where the address is calculated as a hash of a 32-byte arbitrary ID and the owner address. Within the chunk content in addition to the data and the 8-byte span as before, there is the ID and a digital signature made by the owner. The signature is signed on the ID and the BMT chunk hash which is the same calculation as the BMT chunk adress above. The single owner chunk can be validated by extracting the ID, signature, and payload. Then by reconstructing the content to sign, with ID and BMT hash, one can get the owner address, which finally is used with the ID to reconstruct the chunk address [20].

3.2.2 Chunk replication

When a node hosting a chunk leaves the network, then the chunk cannot be retrieved. In order to combat this situation the chunk is also stored by the nodes nearest neighbours. The number of times a chunk should be replicated, is called the **redundancy factor** r [20]. Currently in Swarm r is set to 4. The nearest neighbours, with neighbourhood depth d, must contain at least r peers. Each node also possesses a cache, where they can keep chunks that have been forwarded to them for quick relay access.

3.2.3 Push & pull syncing

Push syncing is the protocol that moves a chunk to its intended neighbourhood. Forwarding Kademlia can be used to deliver the chunk to the closest node to its address. A receipt of storage is then passed back from that node.

Pull syncing makes sure that each node in the neighbourhood have their chunk storage synchronized.

3.3 Storage incentives

Storage incentives are ways of financially compensating individuals for providing their storage capacity. The storage incentives of Swarm are handled on-chain, by use of Ethereum based smart contracts, currently deployed on the Gnosis chain [24]. The smart contracts are called: PostageStamp, PriceOracle, Staking, and Redistribution. The latter will be covered separately in section 4.2.

3.3.1 Postage stamps

The funding for the storage incentives come from when clients pay to upload contents, the way they do this is by purchasing **postage stamps**. The postage stamp is associated with a chunk, and the digital signature of the uploader. It acts as a proof of payment for the chunk. Nodes are more likely to keep storing a chunk with a high value associated postage stamp.

Buying a stamp for each 4KB chunk is a tedious process, so instead they are paid for in batches. It is the batch depth which decides how many chunks a postage stamp can pay for, this number is a base 2 logarithm. With batch depth of 8, it pays for $2^8 = 256$ chunks. A per-chunk balance is kept, which is the total amount of BZZ paid for the batch, divided by the number of chunks paid for. Importantly anyone can choose at a later date to increase the batch balance, although only the owner can increase the batch depth to dilute the price. This means that data people deem important can remain on the service, even after the owner has lost interest. Batches expire when their balance runs out, and when this happens they are no longer protected against being evicted [4].

Postage batches are handled in the PostageStamp smart-contract. Additionally it keeps track of the pot of rewards to be redistributed. The redistribution itself happens in rounds, orchestrated by the Redistribution contract. We will discuss this in detail in section 4.2.

3.3.2 Depths

The storage incentives have some key metrics for describing the data stored on the network, these are the depths, and there are three in total. First the **reserve depth**, which tells us about how much storage has been reserved by the purchase of postage batches. It is calculated by using the total number of storage slots in valid postage batches, the reserve size. And taking the base 2 logarithm of that reserve size, rounding it upwards towards the nearest integer[20].

Secondly the **storage depth**, which is important to the operation of the redistribution contract as it defines the number of neighbourhoods. Storage depth is a measure of the postage batches that are currently utilized to store data, as in the total number of chunks uploaded to the network. The storage depth of a node is the lowest PO at which the node stores all batch bins[20].

The third depth is the **Neighbourhood depth**, it is somewhat similar to the same depth defined for the overlay network. It relates to the local replication of chunks, each neighbourhood decided by the storage depth should have at least four nodes (redundancy factor r). The neighbourhood depth of a node is the highest PO: d, that allows the address range decided by the d bit address prefix, to contain at least three (r-1) other peers [20].

3.3.3 Price oracle

The PriceOracle contract is a measure to self regulate the price of storage. It calculates the current **storage rent**. At each block in the blockchain, the balance for the postage batches decrease by the amount of the storage rent [4]. It is the accumulated total storage rent that becomes the reward pot mentioned earlier. The price regulation is a result of feedback from the Redistribution contract. The Redistribution contract at the end of a round tells the PriceOracle contract how many neighbourhood nodes participated honestly. If this amount is equal to the redundancy factor, then the storage rent remains the same. Otherwise if it is larger then the rent decreases to encourage more files uploaded to the network. Final case if it is less than the redundancy factor, the storage rent increases to increase the number of storage nodes[24].

3.3.4 Staking

Swarm uses a Proof of Stake (PoS) system for storage nodes, the idea is to make sure that only invested/serious node operators join the network. Each node has a chance to earn storage incentives proportional to their staked amount of BZZ. Furthermore this provides each node with some accountability for their actions, since it is possible to both freeze and/or slash a node's stake if they misbehave. Having one's stake frozen means being ineligible to take part in the storage incentive. This is also the case if one's stake is slashed below the minimum BZZ needed to be staked. These actions, and the management of stake is handled in the Staking contract.

The full overview of the storage incentive system can be seen in figure 3.1. As a brief summary clients pay for chunks in postage batches in the PostageStamp contract. The storage rent is deducted from the postage batches, and added to a pot, that is to be redistributed. Storage nodes stake BZZ through the staking contract in order to be eligible to participate in the redistribution. Misbehaving nodes can have their stake frozen, and/or slashed. The Redistribution contract hands out the pot from the PostageStamp contract, and gives feedback to the Staking, and PriceOracle contract. And the PriceOracle contract adjusts the price of the storage rent based on said feedback.

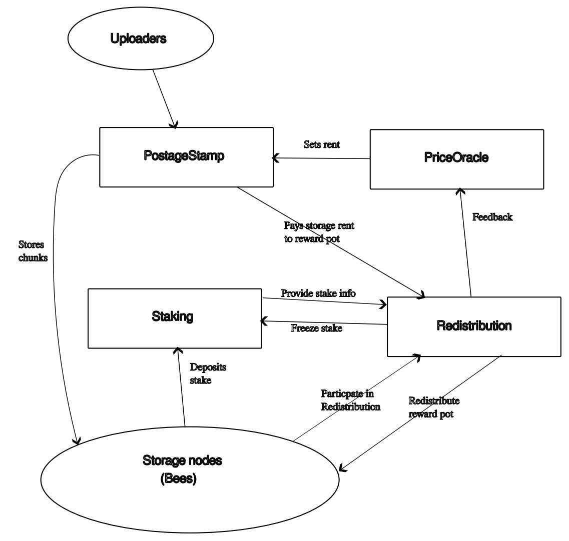


Figure 3.1: Block diagram of the storage incentive system

Chapter 4

Approach

4.1 Introduction

In this chapter we will be analysing the existing redistribution contract by Swarm. While considering the results of said analysis, we define the properties and requirements, that an improved redistribution should have. Additionally discussing the different approaches to take in order to safely and fairly, reward Swarm storage nodes. Subsequently we propose the alpha, and bank solutions. Finally we discuss and analyse what to do with the new bank reward, that comes as a consequence of the proposed solutions.

4.2 Existing Approach

The current redistribution contract acts as a Redistribution Schelling game. A Schelling game is in game theory a coordination game that revolves around a focal point. The idea being that if multiple parties are asked to make decision or find a solution, without being able to communicate, then this focal point would come up naturally and independently as a solution [25]. The coordinators in this case being a neighbourhood of storage nodes, and the focal point the chunks they all store.

Every 15 minutes a round of this redistribution Schelling game occurs. The game runs in three phases, commit, reveal, and claim. First a swarm neighbourhood is randomly selected. Then the commit phase starts: nodes in that neighbourhood should be storing the same data. To prove that the nodes have consensus about the chunks they store, then they must show evidence called Proof of Entitlement (PoE). This evidence needs to be unstealable, such that a node not doing storage does not get a hold of it. That is why a commit/reveal scheme is necessary and hence why it is the commit phase. Furthermore in the commit phase the staking contract is checked to make sure that the node has the minimum amount of staked BZZ to participate.

In the reveal phase the committed PoE are revealed, they submit a "transaction containing their reserve commitments, their storage depth, their overlay address, and the key they used to obfuscate the commit". The reserve commitment being a hash representing the stored chunks, it acts as PoE in this case. Checks are done to verify that the sent information does hash to the previous committed message, and that the node is in the storage depth that they claim to be in.

The winner is decided in the claim phase. To decide the winner of the game one reveal is selected as the truth. Based on this truth the participants are divided into roles:

- Honest are those that agree with the reveal
- Liars are the ones who disagree with the reveal
- Saboteurs are the ones that did not reveal or who gave a faulty reveal

One randomly selected winner among the honest nodes can issue a claim transaction on the smart-contract. The winner in a neighbourhood is selected proportionally to their stake. Liars and saboteurs are punished by being frozen. Which means they cannot participate in upcoming rounds [4].

There is an issue that arises since selecting the truth is done at random on the valid reveals. The truth can be a spurious solution that does not correctly correlate with the chunks being stored, as long as it passes the aforementioned checks. A malicious actor could create a PoE that only he knows, and if it is chosen as the truth, then the malicious actor gets all the rewards. But since everyone else also gets frozen in this case, if his neighbourhood is selected again for the reward distribution game, then he can get the rewards for that one as well as the only participant. However there is still a significant risk of becoming a liar himself and getting frozen. This is not the only reason a node would select a different truth than the others however. For instance it might be running an earlier version of swarm. Or its chunk storage could be not synced somehow, maybe due to network instability.

Looking at transaction data from Swarmscan [14] using readSi script from ethersphere/bee-scripts [26], for 3144 rounds in January-February 2024. There have been 419 times that a minority revealed truth has been selected, which is 13% of the rounds. Furthermore different revealed values are present in 60% of the rounds.

4.3 Analysis

In this analysis we will analyse the claim phase of the current redistribution contract, to see what the real impact that the issue with the freezing mechanism has. We define the expected reward for Storage nodes in Swarm, first with everyone following the protocol honestly. And secondly with the case where there is one malicious node in each neighbourhood. We show that for this second case, that the malicious node can expect an equal expected reward to the honest nodes in the neighbourhood.

4.3.1 Expected honest node reward

First of all what will be the expected rewards in a world where everyone does as they are supposed to? In this case the selection of revealed truth is always going to result in every node being able to participate, therefore we only need to look at an individual node's chance of being selected as a winner. The expected reward for any sinlge $node_k$ in a round is the probability for it to be selected given that its neighbourhood is selected (eq. 4.1). We assume that the winnings W_i for each round i, is equal to 1. Therefore we can simplify the expected distribution of rewards after n rounds to be equal to equation 4.2. Let $S(hood_k)$ be the sum of stake in a neighbourhood where $node_K$ is a member, and $S(node_k)$, be the stake of $node_K$. Finally let $nhood$ be the number of neighbourhoods.

$$E(reward) = \prod_{i=1}^{n} P(hood)P(k|hood)W_i \tag{4.1}$$

$$E(reward) = \prod_{i=1}^{n} \frac{1}{nhood}\frac{S(node_k)}{S(hood_k)} \tag{4.2}$$

From equation 4.2, we have that the expected reward is exactly proportional to a node's stake.

4.3.2 One malicious node

We now introduce one malicious node into each neighbourhood. For each redistribution round we have two different outcomes; whether the honest nodes win, or the malicious node wins. Nodes will be frozen in each round, making subsequent rounds affected by who was frozen in the previous round. We therefore have that the expected reward of round n is dependent on what happened in the previous rounds. Every time a round is ran the set of frozen rounds is updated, and the chance of winning in the next round for any node depends on that update. We consider this as the outcome of the next round only being dependent on the frozen state of the previous round. This gives us a Markov property, which allows us to model the situation as a Markov chain model.

A visualization of our Markov chain model can be seen in figure 4.1. The model has three states $s_1, s_2, s_3 = unfrozen, mfrozen, hfrozen$. In the $unfrozen$ state, no node is frozen. In the $mfrozen$ state, the malicious or faulty node is frozen, whereas in the $hfrozen$ state all the honest nodes are frozen. Continuing on we will be defining the transition probabilities: $P(h)$, $P(m)$, and $P(uf)$.

First let us take a step back to redefine the expected reward for honest nodes. Let us assume that stakes among nodes are evenly distributed. The equal stake assumption can be justified by considering that currently most nodes only stake the minimum amount. However we mainly think that since the assumption that the reward is proportional to stake holds throughout the process, which is backed in Tjessem's work [16]. Nodes will ever only have an equal chance of winning with the same stake. For reference the honest node equation 4.2 of the previous subsection can now be written like in equation 4.3.

$$E(reward) = \prod_{i=1}^{n} \frac{1}{nhood} \frac{1}{|k\epsilon hood|} \tag{4.3}$$

Where n is the number of rounds, nhood is the number of neighbourhoods, and hood is a set of nodes k representing the selected neighbourhood.

$$E(reward) = \prod_{i=1}^{n} \frac{r}{N} \frac{1}{r} = \frac{1}{N} \tag{4.4}$$

Let us also assume that the number of nodes in each neighbourhood is uniform, and equal to the redundancy factor r. The number of neighbourhoods $nhood$ can then be described as $\frac{N}{r}$, where N is the total number of nodes. This results in the expected honest reward found in equation 4.4.

Probability of a malicious node having its reveal selected as the truth is $P(m)$, and the probability for the revealed truth to be the honest reveal is $P(h)$.

$$P(m) = \frac{1}{nhood} \frac{1}{r} = \frac{1}{N} \tag{4.5}$$

$$P(h) = \frac{1}{nhood} \frac{(r-1)}{r} = \frac{(r-1)}{N} \tag{4.6}$$

The probability of leaving the $unfrozen$ state, and going to either $mfrozen$ or $hfrozen$, are $P(h)$, and $P(m)$ respectively. And the probability of staying $unfrozen$ is $1 - (P(m) + P(h))$.

We now need to find the probability of being unfrozen $P(uf)$. Each node in practice is frozen for a set number of rounds. However, in a Markov Chain model we need to use probabilities, since each row in the transition matrix have to sum up to 1. For said probability we say that we expect a frozen node to unfreeze after a certain number of rounds. We define ft, freezing time, as the number of rounds that a node is frozen. Which gives us $P(uf) = \frac{1}{ft}$. As an additional consequence of using a Markov chain model we have the probability of remaining frozen, which will be $1 - P(uf)$. Although the previously deterministic unfreezing process, is now nondeterministic. It will still closely approximate the deterministic behaviour in the long run.

$$ft = penaltyMultiplierDisagreement * roundLength * 2^{truthRevealedDepth} \tag{4.7}$$

In the current redistribution contract the freezing time ft is calculated like shown in equation 4.7 [24]. The penaltyMultiplierDisagreement is a tune-able parameter for how severe we want to punish nodes disagreeing with the revealed truth. Currently this penalty is set to 1 in the contract, and as such does not impact the time at all. The Markov chain model calculates round by round, and as such the roundLength also gets set to 1 in the model. This leaves the simulation freeze time as $ft_{sim} =$

$2^{truthRevealedDepth}$, truthRevealedDepth being the storage depth revealed in the truth. $2^{storageDepth}$ calculates the number of neighbourhoods, $nhood$, which we previously defined as N/r. Substituting N/r into equation $P(uf) = \frac{1}{ft}$, leaves us with equation 4.8.

$$P(uf) = \frac{1}{nhood} = \frac{r}{N} \tag{4.8}$$

Putting it all together, we have the complete Markov chain transition matrix in (tab. 4.1). And as previously mentioned there is a visualization of the Markov chain in figure 4.1.

	unfrozen	mfrozen	hfrozen
unfrozen	1 – (P (m) + P (h))	P(h)	P(m)
mfrozen	P(uf)	1 - P(uf)	0
hfrozen	P(uf)	0	1-P(uf)

Table 4.1: Transition matrix

The expected rewards of each singular node are dependant on the probabilities of being in each state, $P(s_1)$, $P(s_2)$, $P(s_3)$ (eq. 4.9). By letting the reward value be equal to 1, it can be omitted from the equations. However, do note that it is still a factor in each term.

$$E(Reward) = P(s_1)P(win|s_1) + P(s_2)P(win|s_2) + P(s_3)P(win|s_3) \tag{4.9}$$

When considering the chances to win for a singular node in the $unfrozen$ (s_1) state, it is uniformly $\frac{1}{N} = P(m)$, regardless of what proof of storage it provides. Where honest and malicious nodes differ however, is their chances of winning in states s_2, s_3. The malicious node will always lose in state s_2, and always win in state s_3 (eq. 4.10). And an honest node will always lose in state s_3. While in state s_2 it will need to compete with the rest of the nodes in the neighbourhood submitting the right proof, which is: $\frac{1}{r-1}$ (eq. 4.11).

$$E(Reward_m) = P(s_1)P(m) + P(s_2) * 0 + P(s_3) * 1 = P(s_1)P(m) + P(s_3) \tag{4.10}$$

$$E(Reward_h) = P(s_1)P(m) + P(s_2) * \frac{1}{r-1} + P(s_3) * 0 = P(s_1)P(m) + P(s_2)\frac{1}{r-1} \tag{4.11}$$

From the above equations we have that $E(Reward_m) = E(Reward_h)$, if $P(s_3) = \frac{P(s_2)}{r-1}$. To find out if that is the case, we have the equations for finding $P(s_1)$, $P(s_2)$, and $P(s_3)$ 4.12, 4.13, and 4.14. These are derived from the incoming edges to

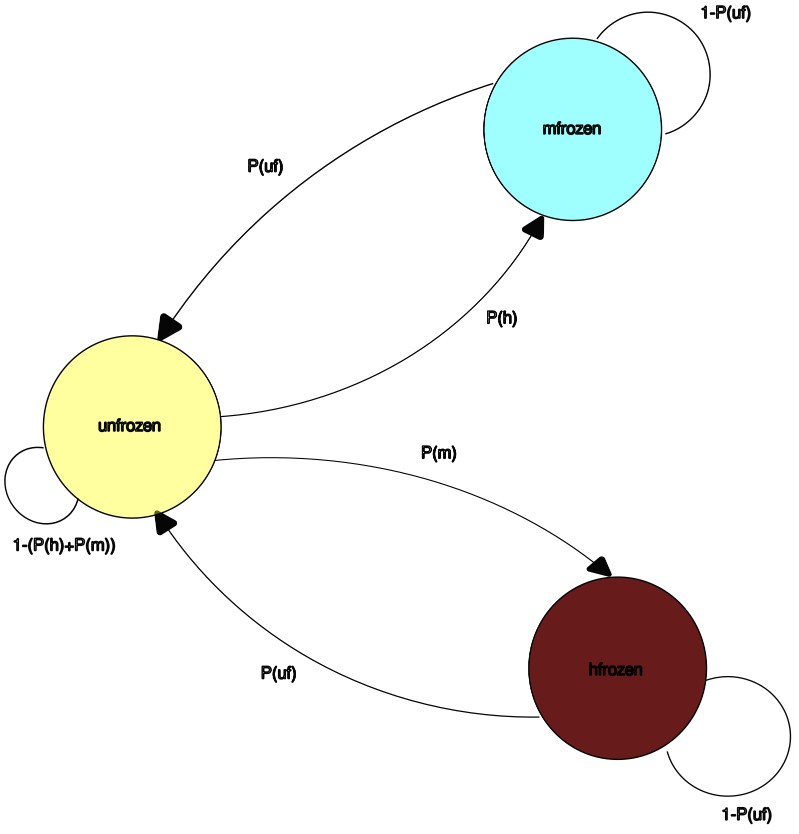


Figure 4.1: The Markov chain model

the target state $P(s_x \rightarrow s_y)$, factored by the probability of being in the initial state $P(s_x)$.

$$P(s_1) = P(s_1 \rightarrow s_1)P(s_1) + P(s_2 \rightarrow s_1)P(s_2) + P(s_3 \rightarrow s_1)P(s_3) \tag{4.12}$$

$$P(s_2) = P(s_2 \rightarrow s_2)P(s_2) + P(s_1 \rightarrow s_2)P(s_1) \tag{4.13}$$

$$P(s_3) = P(s_3 \rightarrow s_3)P(s_3) + P(s_1 \rightarrow s_3)P(s_1) \tag{4.14}$$

Solve equation 4.13 for $P(s_1)$:

$$P(s_2) = (1 - \frac{r}{N})P(s_2) + (\frac{r-1}{N})P(s_1) \tag{4.15}$$

$$P(s_1) = \frac{NP(s_3) - (N-r)P(s_r)}{r-1} = \frac{rP(s_3)}{r-1} \tag{4.16}$$

And also solve equation 4.14 for $P(s_1)$:

$$P(s_3) = (1 - \frac{r}{N})P(s_3) + \frac{1}{N}P(s_1) \tag{4.17}$$

$$P(s_1) = NP(s_2) - (n-r)P(s_2) = rP(s_2) \tag{4.18}$$

Finally substitute eq. 4.18 into eq. 4.16.

$$rP(s_3) = \frac{rP(s_2)}{r-1} \tag{4.19}$$

$$P(s_3) = \frac{P(s_2)}{r-1} \tag{4.20}$$

We have know shown that since $P(s_3) = \frac{P(s_2)}{r-1}$, that the expected reward for honest nodes $E(Reward_m)$, is equal to the expected reward for the malicious node $E(Reward_h)$.

It can be tested out experimentally by assigning values to N, and r. Noted that the current value for r in Swarm is 4, we should still try different values. $P(s_1)$, $P(s_2)$, and $P(s_3)$ can be calculated by finding the left eigenvectors of the transition matrix. Whichever one of the eigenvectors has the eigenvalue 1, corresponds to the eigenvector holding the information for the state probabilities. By normalising that eigenvector by dividing each element with the sum of all the elements in the vector, the percentages are found.

We also simulated the single malicious node scenario, by modifying Kristian H. Tjessem's SwarmSI [16]. In our modification we built on the FixedIdealSwarmNetwork, in this network all the neighbourhoods have the same number of nodes equal to r. We made it possible for nodes to submit different truths, and for the network

N	R	s_1	s_2	s_3	$\frac{P(s_2)}{r-1}$
512	4	0.5	0.375	0.125	0.125
2048	4	0.5	0.375	0.125	0.125
7777	5	0.5	0.400	0.100	0.100
4096	8	0.5	0.437	0.062	0.062

Table 4.2: Markov chain experimental results

to handle freezing, and unfreezing of nodes. And for the SelectWinner method to account for frozen nodes.

We ran a single run of the simulator with $r = 4$, and 1024 nodes, for 10000 rounds. Let h be the group of honest nodes, and m the group of malicious nodes. The results for each group can be seen in table 4.3. We took the earnings for the nodes at the last round of the simulation, and found the average earnings: $E(h)$, $E(m)$, and the standard deviation: $SD(h)$, $SD(m)$.

Group	h	m
Mean	0.000988	0.000941
SD	0.000350	0.000410
N	768	256

Table 4.3: SwarmSI result

On the simulation data we did an unpaired t test, in order to check if any significant difference between the two groups could be found. With $\alpha = 0.05$, $t = 1.7799$ and $df = 1022$. The 95% confidence interval found goes from -0.00000488 to 0.0000988. And the P value is 0.0754. This P value is not quite statistically significant, therefore the SwarmSI simulation also concludes with $E(h) = E(m)$. Admittedly this particular test, with only this result, is weak on its own. However in combination with the above mathematical proof, We have deemed it sufficient for now.

What our analysis in this section shows is that the expected reward (proportional to stake) for revealing the same truth as other neighbourhood nodes, and for revealing a different truth to that, is the same. In game theory terms the expected honest strategy is only a weakly dominating Nash equilibrium. Which means that if one is following that strategy, then there is no utility gained for switching to the alternative, but there is also no reason not to. In fact additional utility could be gained by the malicious node by not using storage capacity to store the neighbourhood chunks. In particular since storage rent adjustments are based on how many nodes are eligible to claim the reward of the round. This could lead to negative consequences for the network as a whole, if this number of eligible nodes is repeatedly marked as 1 due to others getting frozen. Since when this number is less than r, the storage rent increases, which gives better terms to storage nodes, and therefore the single claimant. If this happens repeatedly one could only speculate that it can increase the storage rent such that uploaders will take their files elsewhere.

4.4 Proposed Solution

In the following section we first set up what requirements the new redistribution contract needs to have. After which we comment on possible approaches for our solution that mitigates the issue of node's sending in arbitrary reserve commitments. Finally we present our solutions the bank, and alpha solution.

4.4.1 Requirements

The properties and considerations our solution should hold are as follows:

Fairness: A storage node gets rewarded proportional to the effort it puts in. In Swarm since every node in a neighbourhood stores the same chunks, that means nodes need to be rewarded in proportion to their stake in the PoS blockchain.

Coordination rewarding: If the storage nodes all decide the same storage depth and proof of reserve capacity. Then they should be rewarded more than if any subset of nodes decide different values.

Sybil resistance: Having multiple storage nodes in the same neighbourhood should not disproportionately increase one's expected reward. Formally given nodes A, B, C, if the stake of C is greater than that of A and B combined. Then for any combination of strategies used by A and B, C can always expect greater or equal rewards.

As the network grows, the storage incentive should remain an enticing protocol, it should be Scalable. Survey by Zahed Benisi et al mentions scalability as one of the greatest challenges for blockchain based systems [19].

The storage incentive process needs to be a high performance solution. Since both hardware and bandwidth usage is not free. Additionally since the process uses Ethereum smart contracts, whatever solution chosen should be as optimised as possible in order for it not to use a lot of gas. The transaction gas overhead makes certain programming approaches that otherwise would be trivial with modern hardware less so. Storing information on chain is more expensive than in memory.

Crucially the property not already taken care of by the existing contract is coordination rewarding. Since with freezing of nodes, if a minority reveal is chosen as the truth, it is expected to gain an equal reward to the others, due to the freezing of the majority. See previous analysis.

The reason for having Sybil resistance is such that someone does not benefit from creating multiple overlays in a neighbourhood with less stake, compared to placing all their stake in one overlay. Storage incentives should be done in a way that incentivizes nodes to join neighbourhoods in a symmetric manner. To fix a previous issue of imbalanced neighbourhoods, the number of nodes in a neighbourhood would vary greatly, Swarm introduced a way to mine the overlay for the storage node. And made an interface highlighting neighbourhoods that would benefit from more members, so new nodes would try join those neighbourhoods[27]. This could imbalance neighbourhoods more if people tried to mine themselves into having more in the same neighbourhood. This behaviour is undesirable since it limits the decentralization of certain chunks. Which is why the incentives should discourage such behaviour. On the contrary an actor having multiple nodes each in a different neighbourhood is a good thing, because it increases storage capacity of the network as a whole.

4.4.2 different approaches

We want to update the existing contract, to further incentivize storage nodes in a neighbourhood to provide the same reserve commitment. Which is an indication that they are all storing the same chunks. This is because in the current contract committing a reserve commitment in the minority is an equal strategy to following the majority.

Improving the existing redistribution contract has two main easily identifiable approaches. The first is to work on a more robust truth reveal. It would simplify the process if the smart-contract can decide on the truth on its own for what chunks store. Additionally more restrictions might be possible to add for what constitutes a valid reveal. The other approach is to live with not being able to 100% know the truth. This is the current approach, and why the truth chosen is left up to chance. If the method of calculating the reserve commitment changes between versions, and one has nodes both on the old version and the new. It might be possible for both versions to take part, if we do not assume a certain truth. Also if a node is acting honest, as in trying to store every chunk the neighbourhood is responsible for, but is unable to obtain all of them either due to network fault, or some other node(s) purposely withholding the chunk [17], it can also have a shot at the reward.

Why Schelling coordination game?

Why not distribute rewards to all storage nodes? If it remains probabilistic, and if the network grows substantially large then being rewarded while being fair in the long run. Might be too much like winning the lottery, and that there is no reasonable security that you will be paid today, or tomorrow. Which could be off putting to some.However since Swarm seeks to have what they call zero cost of entry, the potential lack of liquidity is not that much of a factor. At the time of writing BZZ is worth $\$0.3926$, so starting to stake is not expensive. Furthermore the round time can be adjusted to give more or less payout chances.

Another reason is for performance. Firstly the contract needs to somehow verify that a storage node is not free riding. Which as stated the current smart contract already has some difficulty figuring out. It would take a lot of messages (bandwidth) and smart contract calculations (CPU/gas) to do this for every node. Using random selection anchors to select a single neighbourhood is a boon for performance, since it means the contract only needs to use for loops that loop through neighbourhood members. And neighbourhoods are incentivized to be a manageable size, to increase the chance for any given node to win. In this way the smart contract uses less gas. Secondly, with the current smart contract, only one claim transaction can be made for each round. And that greatly reduces the amount of transactions that need to be stored on chain.

4.4.3 Redesign options

The reason why the two strategies presented in section 4.3.2 are equal, is because when a single node remains unfrozen, it gets the whole reward. If instead that reward was reduced, the outcome could be different. The expected reward is reliant on two things: the probability of being rewarded, and the value of the reward itself. Thus we can either decrease the chance to win, or hand out less reward. However regardless of which we choose to reduce, there is no distinction outside of how we choose to implement the redesigned smart contract The question is: based on what, do we tweak the expected reward?

Already the redistribution contract sends the number of ”truthy” participants to the PriceOracle contract, which in turn gets compared to r, and ends up regulating the number of nodes in the neighbourhood. We can punish nodes by withholding some reward if the number of truthy participants is less than r. However that is the opposite incentive of what the PriceOracle intends to do in this case. It increases storage rent, to increase number of nodes in the neighbourhood.

Other ways to tweak the expected reward that is more in line with Swarm’s other incentives are as follows:

Number of different revealed values: Ideally we want all storage nodes to decide on the same reveal value, indicating that they are storing the same chunks. Let us say that every node in the neighbourhood gets punished in proportion to number of different reveals. Then as a whole the neighbourhood would get less reward, compared to neighbourhoods with fewer revealed values. But the expected reward within a neighbourhood would still be the same for any chosen reveal.

Number of reveals different from my reveal:

For every node with its own reveal value count the number of reveals from the other nodes in the round with a different reveal value. As an example let 5 nodes reveal the majority reveal, two nodes have the same minority reveal, and a final eight node has its own unique reveal.. Then for every node following the majority there would be 3 different reveals, the two nodes with the same minority reveal have 6 different reveals, and the remaining node would have 7 different reveals. Instead of punishing every node equally based on the number of different reveals, as with number of different revealed values. This allows the node’s voting for the majority reveal, with the lower number of different reveals to be punished less, than those following a minority reveal. Furthermore allowing for coordination rewarding as any nodes with the same reveal are punished less than, nodes with a unique reveal value. However, without involving node's stake we do not have sybil resistance. Additionally this criterion is expensive to implement in practice.

Stake belonging to my reveal: Let rvl_i be the reveal of $node_i$, and $S(rvl_i)$ be the sum of stake, staked by the nodes revealing rvl_i. Utilizing stake also in our solution is a good way to ensure we do not weaken the redistribution contract's Sybil resistance. Using this criterion therefore allows us sybil resistance while it also can allow for coordination rewarding.

In essence all of these methods are ways of achieving the coordination rewarding property. However the ones not taking stake into account: number of different revealed values, and number of reveals different from my reveal, have issues with Sybil resistance. Although the latter could be transformed to be stake belonging to different reveals. Due to the lower complexity the one we are most in favour of utilizing is the stake belonging to my reveal.

The question that arises is what to do with the part of the reward that does not get handed out:

Burn it: The tokens can be burnt, in which case the system loses that part of the reward. One would think that this approach can lead to deflation, benefiting those with a stockpile of tokens.

Add it back to the pot: It can be added to the pot next round, in which case if a faulty node managed to freeze the others. Then it would still be possible to earn some extra reward if its neighbourhood was picked once again in a subsequent round. The natural solution for that is to only inject the leftover reward into the reward pot of a round, after which all frozen nodes in the initial round has been unfrozen.

Bank The money can be claimed by a bank entity, and repurposed for some other incentive.

We can also decide whether to keep freezing mechanism or not. It might be either too complex, or too punishing to have both the proposed new mechanisms and the old freezing. On the other hand since these mechanisms arent mutually exclusive it is entirely possible to combine them. We will be removing freezing in the updated contract however, since it lowers analytical complexity and gas costs.

Now finally we can consider how to punish misbehaving nodes. The main idea is that everyone in the neighbourhood in the current round, gets their chance of winning reduced if there are multiple different reveals. However, it is also possible to punish the neighbourhood in future rounds. By reducing the chance that the neighbourhood gets selected in the future. Perhaps as an additional measure if the neighbourhood repeatedly misbehaves by having many unique reveals.

4.4.4 Chosen solutions

So far we have spent some time discussing the existing contract and the possible options that are available for us to adjust, and hopefully improve it. Also we have discussed what properties it should have, and which method possesses which properties. In this section we will present the concrete solutions we are going forward with.

What we have come up with are two solutions that use the **stake belonging to my reveal**, criterion. Let s_i be the stake, and v_i be the revealed value of $node_i$. Also let Φ be the set of all reveal values v in the redistribution round. Finally let $S(v)$, the stake belonging to each reveal value, be the sum of all s_i where $v_i = v$, shown in equation 4.21.

$$S(v) = \sum_{v_i \epsilon \Phi, v_i = v} s_i \tag{4.21}$$

The first of these two solutions, is the one alpha solution, due to the use of the Greek letter α to act as a scaling parameter. The second is the Bank solution, named after the bank player we introduce.

We have chosen these two different solutions because we found that the alpha solution is difficult to implement in the solidity programming language while being economic in terms of gas cost. While the bank solution is easier to implement, but is not as intuitively coordination rewarding. The alpha solution gives each storage node a chance to win relative to its stake and the stake committed to that storage node's reveal value. For example if the storage nodes are split by running different versions of the Swarm bee client, each calculating a different reveal value, then the alpha solution lets each respective version cooperate for an increased chance to win the round. Whereas the stake for the bank player in the bank solution adjusts itself only on the highest $S(v)$. Which means no inherent cooperation between the majority not staking on the highest $S(v)$.

Alpha

In this solution, we calculate the stake belonging to each reserve proof reveal,

The chance for a node in a selected neighbourhood to win, is given in equation 4.22. $S(\Phi)$ is the stake belonging to all reveals, or the total amount of stake in the redistribution round. Equation 4.22 gives each node a chance to win proportional to their share of the stake in the first factor. In the second factor: $\frac{S(v_i))^\alpha}{S(\Phi)^\alpha}$, the reward is made proportional to the stake belonging to the node's reveal.

$$R(node_i) = \frac{s_i}{S(\Phi)} \frac{S(v_i))^\alpha}{S(\Phi)^\alpha} \tag{4.22}$$

If there is stake attached to different reveal values in a round, then this second factor will make it so each node has a reduced chance to win depending on which reveal value they have. It also means that a storage node is not guaranteed to win in a round, as the sum of all winning probabilities for storage nodes does not equal to one. When no storage node is the winner, we say that the bank wins.

There is a scaling parameter α, $\alpha\epsilon(0, 1]$, that is meant to offset the severity of punishment for when a node's stake is small. With $\alpha = 1$, nodes with less stake are punished unreasonably much, compared to those with a lot of stake. Although this can be a desirable property to deter nodes acting on their own. In general the lower α is the less impact the $\frac{S(v_i))^{\alpha}}{S(\Phi)^{\alpha}}$ factor has. Reducing the equation to the first factor $\frac{s_i}{S(\Phi)}$.

Bank

For the bank solution we introduce the bank as a player in the Redistribution game just like the nodes. The idea is to control the bank's chance of winning as the same thing as storage nodes going unrewarded. The bank nodes stake is defined in equation 4.23, it is the sum of all stake minus the reveal value with the highest stake attached to it.

$$s_{bank} = sum(S(v)) - max(S(v)) \tag{4.23}$$

The banks stake will be added to the total stake, this means that the bank has an equal chance of winning as all the other reveal values not equal to $max(S(v))$. It is possible to tune the bank solution as well by adjusting the stake the bank receives. As the bank solution does not disproportionately punish nodes with small stake like the alpha solution. And in the interest of keeping it simple, we are not tuning parameters to the bank solution. With that the game is played normally but with an extra bank player (eq 4.24).

$$R(node_i) = \frac{s_i}{S(\Phi) + s_{bank}} \tag{4.24}$$

Example

Here is an example scenario that highlights the differences between the solutions. Furthermore we will be using this example to later test our implementation.

There are three different reveals, r_1, r_2, and r_3, each with 0.5, 0.4, and 0.1 ratio of the stake respectively. In the bank solution it is obvious what we mean by the bank player winning, but we will here also consider the case where no storage nodes win the round as the bank winning with the alpha approach. In table 4.4, the expected winnings of following each reveal, for the respective incentive contract and α value has been noted down.

Having at least three different values is important to highlight the difference between the solutions, as with only two stake fraction camps the solutions work largely the same. The bank stake in the bank solution is set to reflect the largest of the stake fractions. And as such each stake fraction is punished collectively based on the majority. While for the alpha solution each stake fraction is punished individually based on their stake. With two stake fractions each value looks to be simply reflective of the majority fraction.

$\frac{s_i}{S(\Phi)}$	r_1 0.500	r_2 0.400	r_3 0.100	bank tbd
bank win %	0.333	0.267	0.067	0.333
$\alpha = 1$ win %	0.250	0.160	0.010	0.600
$\alpha = \frac{1}{2}$ win %	0.350	0.250	0.030	0.370
$\alpha = \frac{1}{3}$ win %	0.400	0.290	0.050	0.260

Table 4.4: Expected relative winnings

Notably if $\alpha = 1$ (no tuning), then the bank wins more than half the time. Furthermore r_1's chances have been cut in half, while r_3 now is ten times less likely to win. This is the issue discussed earlier in subsection 4.4.4, where the node with the lesser stake gets punished disproportionately. We believe that the punishment in this situation to be too harsh, and therefore the need for α to scale the punishment. As one can see in table 4.4 decreasing α evens it out.

It is hard to say with precision what is the correct punishment however. We need to consider both how often the neighbourhood as a whole should lose, and how fairly each stake fraction is treated. For the former we have in this case that half of the stake is not belonging to r_1, and as such it is best if the bank win percentage does not exceed 0.5. This factor can be seen as how live the neighbourhood is, as in if can expect that eventually a storage node will win. For the latter the bank solution has punishes each stake fraction uniformly, while the alpha solution punishes smaller stake fractions more than the larger ones. This property of the alpha solution can be desirable since it means nodes are rewarded for coordinating towards bigger stake fractions. At the same time we do not know the true reveal value, so we do not want this property to be too prominent. Letting α be 0.333 is the best we have in this regard.

4.5 Bank rewards: keep or burn?

The existing redistribution contract has every round pay out rewards to a winner, where the reward pot for that round is taken from the storage rent due from all of the postage batches. But we have now introduced a chance for no storage node to win, and this ultimately leaves a lot to consider regardless of what we choose to do with the reward. This section is wholly dedicated to discussing the potential issues of each of the approaches mentioned before: burn, keep it in the reward pot, or give it to a bank entity.

4.5.1 Effects of carrying over pot in the next round

In this section the impact of keeping the pot for the next round when the bank wins will be discussed analytically. The reason why we do this is to make sure we cover our ground, and do not introduce worse exploits than the one we are looking to mitigate. It is thinkable that with nodes in multiple neighbourhoods an operator can find a way to increase their expected reward by trying to win not only the reward for the current round, but for the prior rounds as well.

We look at two cases of how a malicious node operator can take advantage of the rewards being carried over. In the first case the operator has the same stake in multiple neighbourhoods. The operator is hoping that by giving a faulty reserve commitment, he can capitalise on being part of many neighbourhoods, in order to eventually earn the carried over reward. We find that this first case strategy for both solutions only provides a higher expected reward, under infeasible conditions. The second case is similar, but we now consider that the operator has two different stakes, each in half of the neighbourhoods that he participates in. In this second case we find that the bank solution is vulnerable, while the alpha solution is more resistant.

Like we did earlier in section 4.3.1 we are going to start this out by assuming that everyone is honest and then introduce a possible malicious actor. And then see if that actor has a possible strategy that is either equal, or better than remaining honest. In the all honest case both bank and alpha approach is the same as before, the expected reward of a node is equal to its stake divided by the total stake of reveals in the round (eq. 4.2). As before all rewards are assumed to be equal to 1.

Bank

We will now begin analysing the scenario where there is one malicious node, starting with the bank solution. Let Φ be the set of reveal values, and $S(\Phi)$ be the sum of stake belonging to all reveal values. Let node x, with stake s_x be a part of k neighbourhoods, and choose to reveal a different commitment hash than the other nodes in each neighbourhood which have $S(\Phi) - s_x$ stake combined. For the sake of convenient notation let pn be the fraction of neighbourhoods node x is in. This gives the bank a chance of winning, which is dependant on whichever of s_x, and $S(\Phi) - s_x$ is the highest. Which gives us two separate cases to solve one where $\frac{s_x}{S(\Phi)} <= 0.5$, and one where $\frac{s_x}{S(\Phi)} >= 0.5$. To start of with we will assume $S(\Phi) - s_x$ is the highest, as in $\frac{s_x}{S(\Phi)} <= 0.5$, which gives a bank stake s_{bank} as seen in equation 4.25.

$$s_{bank} = S(\Phi) - (S(\Phi) - s_x) = s_x \tag{4.25}$$

Which means both the bank, and the malicious node will have the same chance of winning $P(M)$, and $P(B)$ respectively as in equation 4.26, given that a neighbourhood with node x has been chosen.

$$P(M) = P(B) = \frac{s_x}{S(\Phi) + s_x} \tag{4.26}$$

Conversely node x's chance of winning if acting as a honest storage node: $P(H)$ is described in equation 4.27.

$$P(H) = \frac{s_x}{S(\Phi)} \tag{4.27}$$

The node x while playing dishonestly either wins in a round with probability $P(M)$, or the bank wins with probablity $P(B)$.

The expected reward in a round for this strategy can be summed up as the chance for x to win as a malicious node plus the chance to win an extra round worth of rewards given that the bank won in previous rounds. The chance that the bank wins in each subsequent round decreases over time, and leaves us with the expected reward shown in equation 4.28. The probability of a neighbourhood with x being selected in each round pn plays a role for each term. We assume pn, $P(M)$, and $P(B)$ remain constant for every round. Let n be any number of rounds.

$$E(x_m) = pn * P(M) + pn * P(M)\sum_{i=1}^{n}(pn * P(B))^i \tag{4.28}$$

What we are interested in is whether $E(x_m) < pn * P(H)$

$$pn * P(M) + pn * P(M)\sum_{i=1}^{n}(pn * P(B))^i \leq pn * P(H) \tag{4.29}$$

$$P(M) + P(M)\sum_{i=1}^{n}(pn * P(M))^i \leq P(H) \tag{4.30}$$

In step seen in equation 4.30, we cancel out pn on each side, and substitute $P(B)$ with $P(M)$, to take advantage of that $P(B) = P(M)$. Let us ignore the right hand side for now and write out the sum on the left hand side.

$$P(M) + pnP(M)^2 + pn^2P(M)^3 + pn^3P(M)^4 \quad ... \quad pn^{n-2}P(M)^{n-1} + pn^{n-1}P(M)^n \tag{4.31}$$

This results in the left hand side being a new geometric series: $\sum_{i=1}^{n} pn^{i-1}P(M)^i$. We are interested in what this series converges to as n approaches infinity, which we can find by using the geometric series formula: $\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-k}$, where $a_1 = P(M)$, and $k = pn * P(M)$ (eq. 4.32).

$$\sum_{i=1}^{\infty} pn^{i-1}P(M)^i = \frac{P(M)}{1-(pn * P(M))} \tag{4.32}$$

Now that we have solved the sum on the left hand side we can bring back $P(H)$ on the right hand side (eq. 4.33).

$$\frac{P(M)}{1-(pn * P(M))} \leq P(H) \tag{4.33}$$

$$\frac{\frac{s_x}{S(\Phi)+s_x}}{1-(pn*(\frac{s_x}{S(\Phi)+s_x}))} \leq \frac{s_x}{S(\Phi)} \tag{4.34}$$

We first consider the edge case where node x exists in every neighbourhood, as in $pn = 1$.

$$\frac{\frac{s_x}{S(\Phi)+s_x}}{1-(1*(\frac{s_x}{S(\Phi)+s_x}))} \leq \frac{s_x}{S(\Phi)} \tag{4.35}$$

$$\frac{\frac{s_x}{S(\Phi)+s_x}}{\frac{S(\Phi)+s_x-s_x}{S(\Phi)+s_x}} \leq \frac{s_x}{S(\Phi)} \tag{4.36}$$

$$\frac{s_x}{S(\Phi)} \leq \frac{s_x}{S(\Phi)} \tag{4.37}$$

Equation 4.37 shows that the left hand side is equal to the right hand side, if $pn = 1$. Furthermore if $pn < 1$, we can see from equation 4.33, that the term $(pn * P(M))$ decreases since both pn, and $P(M)$ are less than one. When $(pn * P(M))$ decreases, the denominator $1-(pn*P(M))$, increases, which ultimately means that the fraction on the left hand side decreases and has its maximum value when $pn = 1$. Thus we can conclude that in this case the malicious strategy is only ever equal to playing honest, if node x is able to join every single Swarm neighbourhood, which is an infeasible task.

However, this was only while $\frac{s_x}{S(\Phi)} <= 0.5$, for completeness sake we will now consider when $\frac{s_x}{S(\Phi)} >= 0.5$. This means redefining $P(M)$ (eq. 4.38), and $P(B)$ (eq. 4.39), while we now have that $P(M) \neq P(B)$.

$$P(M) = \frac{s_x}{2S(\Phi)-s_x} \tag{4.38}$$

$$P(B) = \frac{S(\Phi)-s_x}{2S(\Phi)-s_x} \tag{4.39}$$

We can this time continue from equation 4.30, however this time we will calculate what the geometric series$\sum_{i=1}^{n}(pn*P(B))^i$ converges to, on its own. Which is: $\sum_{i=1}^{\infty}(pn*P(B))^i = \frac{pn*P(B)}{1-(pn*P(B))}$, and that leaves us with equation 4.40.

$$P(M) + P(M)\frac{pn*P(B)}{1-(pn*P(B))} \leq P(H) \tag{4.40}$$

In equation 4.41 we substitute in $P(M)$, $P(B)$, and $P(H)$.

$$\frac{s_x}{2S(\Phi)-s_x} + \frac{s_x}{2S(\Phi)-s_x}\frac{pn*\frac{S(\Phi)-s_x}{2S(\Phi)-s_x}}{1-(pn*\frac{S(\Phi)-s_x}{2S(\Phi)-s_x})} \leq \frac{s_x}{S(\Phi)} \tag{4.41}$$

To simplify equation 4.41 further let $S(\Phi) = 1$, and $0.5 \leq s_x \leq 1$.

$$\frac{s_x}{2-s_x}(1+\frac{pn*\frac{1-s_x}{2-s_x}}{1-pn*\frac{1-s_x}{2-s_x})}) \leq s_x \tag{4.42}$$

$$\frac{s_x}{2-s_x}(1+\frac{pn*(1-s_x)}{2-pn}) \leq s_x \tag{4.43}$$

As before let us check the special case where $pn = 1$, this gives us equation 4.44.

$$\frac{s_x}{2-s_x}(2-s_x)) \leq s_x \tag{4.44}$$

$$s_x \leq s_x \tag{4.45}$$

Which shows that $s_x \leq s_x$, and therefore we have as before that when $pn = 1$ playing maliciously is equal to playing honest, but not greater. If we lower pn then in equation 4.43, the denominator $\frac{pn*(1-s_x)}{2-pn}$ becomes higher, while the numerator gets lower, both of which reduce the effectiveness of the malicious strategy. We can therefore conclude that the bank solution is safe for all s_x, and while the strategies do give equal rewards when $pn = 1$, the additional stake cost of joining every neighbourhood makes it not worth going for.

Alpha

We have now covered how the carried over reward pot affects the bank solution, if an actor possesses stake in multiple neighbourhoods. The alpha solution however will not be affected entirely in the same way, and as such it is important that we redo our analysis for the alpha solution. In this analysis we find that while not particularly feasible, that the alpha solution can potentially be vulnerable to this exploit. Depending on both stake in the neighbourhoods, and the fraction of neighbourhoods the malicious actor is part of. The actor does however, need at least half of the stake in each neighbourhood to be successful.

First of we need to once again redefine $P(M)$ (eq. 4.46), and $P(B)$ (eq. 4.47), while $P(H)$ remains the same. This time $P(M)$ is not equal to $P(B)$, but they are related.

$$P(M) = \frac{s_x}{S(\Phi)}\frac{s_x^\alpha}{S(\Phi)^\alpha} \tag{4.46}$$

$$P(B) = 1-(P(M)+(\frac{S(\Phi)-s_x}{S(\Phi)}\frac{(S(\Phi)-s_x)^\alpha}{S(\Phi)^\alpha})) \tag{4.47}$$

For simplicity's sake let $S(\Phi) = 1$, and $s_x < 1$. This lets us write $P(M)$ as $P(M) = s_x^{\alpha+1}$, and $P(B)$ as $P(B) = 1 - s_x^{\alpha+1} - (1-s_x)^{\alpha+1}$ We can start this time from the same point as equation 4.40.

$$P(M) + P(M)\frac{pn*P(B)}{1-(pn*P(B))} \leq P(H) \tag{4.48}$$

$$s_x^{\alpha+1} + s_x^{\alpha+1}\frac{pn*1-s_x^{\alpha+1}-(1-s_x)^{\alpha+1}}{1-(pn*1-s_x^{\alpha+1}-(1-s_x)^{\alpha+1})} \leq s_x \tag{4.49}$$

In equation 4.49, we substitute $P(M)$, $P(B)$, and $P(H)$ out of the equation. Equation 4.50 is a more compact way to write 4.49 courtesy of wolfram alpha [28].

$$\frac{s_x^{\alpha+1}}{pn(s_x^{\alpha+1} - s_x(1-s_x)^\alpha + (1-s_x)^\alpha - 1) + 1} \leq s_x \tag{4.50}$$

We know trivially that the left hand side is equal to the right hand side when s_x is equal to 0 or 1. We will consider the α values: $\{1, \frac{1}{2}, \frac{1}{3}\}$. Starting with $\alpha = 1$ the inequality becomes what is shown in 4.51. Which we solve to find a correlation between pn and s_x, again with some help from wolfram alpha to refactor the expression [28]. The good part of this correlation is that when $pn = 1$, you still need at least half of the stake in each neighbourhood: $s_x = \frac{1}{2}$, in order to play equally to an honest node. Furthermore if $pn < \frac{1}{2}$, then no matter how much stake the malicious node has, then the expected reward is less, only ever being equal if $pn = \frac{1}{2}$, and $s_x = 1$. On the other hand with $pn = 1$, any $s_x > 0.5$, also means that node x has a much greater reward than other nodes. Although x becomes the majority in this case, and we have assumed $s_x <= 0.5$ for this analysis, what this highlights is the increased coordination rewarding of the alpha solution. Which ultimately makes alpha vulnerable to being exploited if any entity can obtain a monopoly of the network.

$$\frac{s_x^2}{2s_x pn(x-1)+1} \leq s_x \tag{4.51}$$

$$\frac{s_x}{2s_x pn(s_x-1)+1} \leq 1 \tag{4.52}$$

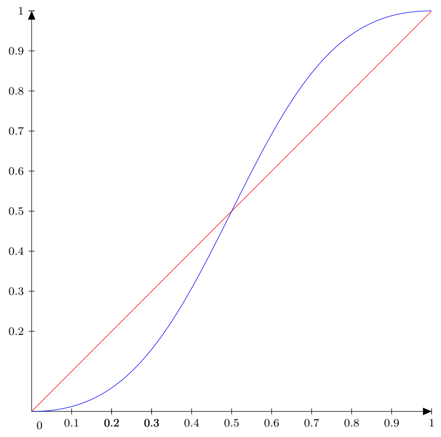
$$s_x \leq 2s_x pn(s_x-1)+1 \tag{4.53}$$

$$\frac{s_x - 1}{(s_x-1)} \leq 2s_x pn \tag{4.54}$$

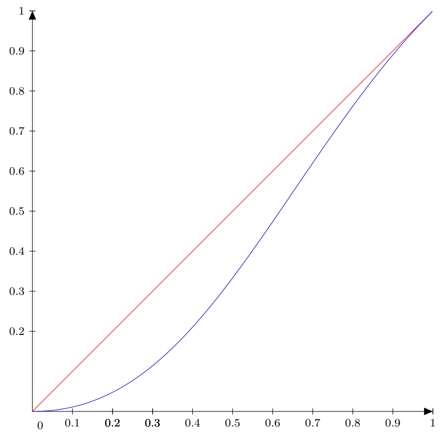
$$pn \leq \frac{1}{2s_x} \tag{4.55}$$

The pn correlation for $\alpha = \frac{1}{2}$, and $\alpha = \frac{1}{3}$, are ultimately too messy expressions to make sense of. However, we can find solutions for specific values of pn, and we know that lowering α makes the left hand side converge towards the right hand side which is s_x. We solve each equation graphically by reading off the intersection between $P(M) + P(M)\sum_{i=1}^{n}(pn * P(B))^i$ and $P(H)$ in the geogebra tool [29]. One can see the solutions for each value in table 4.5. A visualization of the problem can be seen in figure 4.2, where $P(M) + P(M)\sum_{i=1}^{n}(pn * P(B))^i$ is plotted in blue, $P(H)$ is plotted in red.

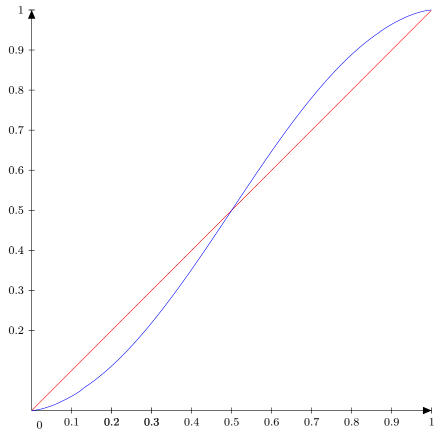
What the results of table 4.5 shows, is that the stake fraction of node x: s_x, generally has to be greater than 0.5 for the malicous strategy to be worthwhile. And the relation found for pn with $\alpha = 1$ is a good approximation for how it is with $\alpha = 0.5$, and $\alpha = 0.33$. However we have found that less stake is needed for the malicious



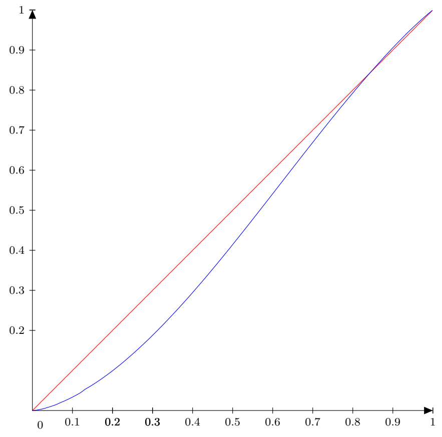
(a) $pn = 1, \alpha = 1$



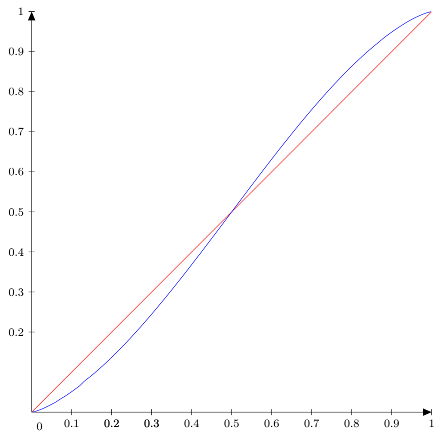
(b) $pn = 0.5, \alpha = 1$



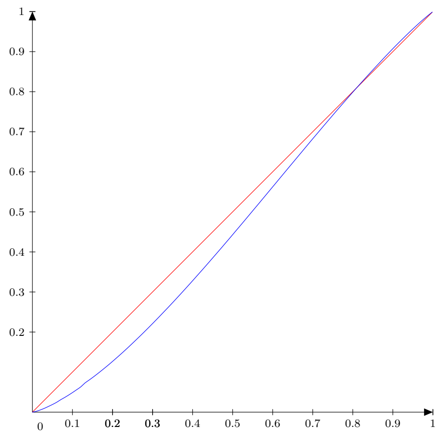
(c) $pn = 1, \alpha = 0.5$



(d) $pn = 0.5, \alpha = 0.5$



(e) $pn = 1, \alpha = 0.33$



(f) $pn = 0.5, \alpha = 0.33$

Figure 4.2: Plots of $P(M) + P(M)\sum_{i=1}^{n}(pn * P(B))^i$ (blue) against $P(H)$ (red), while varying pn and α. x axis represents s_x, while the y axis is probability of winning $P(win)$.

$\alpha = 0.5$		$\alpha = 0.33$	
pn	solution	pn	solution
1	$s_x \leq 0.5$	1	$s_x \leq 0.5$
0.75	$s_x \leq 0.63$	0.75	$s_x \leq 0.62$
0.5	$s_x \leq 0.84$	0.5	$s_x \leq 0.8$
0.25	$s_x \leq 1$	0.25	$s_x \leq 1$

Table 4.5: Solutions for selected pn and α values

node to profit when $pn = 0.5$, the increased odds beyond $s_x \approx 0.8$ are very marginal, but even so the strategies are equal slightly earlier, this point is illustrated in figure 4.2.

The conclusion is still that getting this much stake in this many neighbourhoods is infeasible. While the bank solution is always safe however no matter the stake, the alpha solution does generally provide greater rewards to node x when $s_x > 0.5$ and $pn > 0.5$. On the other hand with the bank the amount of stake for node x s_x, does not matter as an equal strategy is always possible with $pn = 1$, but the malicious strategy does not ever become higher rewarded as it does for alpha. Depending on the α, the alpha solution is skewed more greatly towards the majority, this is related to the alpha parameter discussion when we first introduced the approach in section 4.4.4.

4.5.2 Different stake distributions

In this section we will be seeing if there is a possible way to game the bank reward carryover system by having a different stake values in different neighbourhoods. We believe that by having neighbourhoods where the operator has higher stake they can more reasonably expect to win the reward in those neighbourhoods. While the lower stake neighbourhoods serve as a way to make the bank win, in order to increase the pot earned by those higher stake neighbourhoods. Although it requires a relatively large amount of micro-management, we find that the bank solution is vulnerable to such an approach, while the alpha solution is not particularly so.

We have a node operator in a fraction of neighbourhoods pn, with stakes s_1, and s_2. In half of the neighbourhoods the operator plays with s_1, and the other half with s_2. For the neighbourhoods with s_2 the operator plays as an honest node with a chance of winning in a round with the appropriate neighbourhood selected is $P(\Omega) = \frac{s_2}{S(\Phi)}$. Furthermore with s_1 the operator plays dishonestly with chance of winning $P(M)$, which gives the bank a chance of winning in a selected round of $P(B)$. Let s_1, s_2 <= 1, and $S(\Phi) = 1$. Also let $s_2 = 2s_1$.

The intention is to see if the node operator gets more expected reward than $pn * P(H) = pn * \frac{s_1}{S(\Phi)}$.

$$f(0) = pn * P(M) + \frac{1}{2}pn^2 * P(B)(P(\Omega) + f(0)) <= pn * P(H) \tag{4.56}$$

$$f(0) = P(M) + \frac{1}{2}pn * P(B)(P(\Omega) + f(0)) <= P(H) \tag{4.57}$$

Lets write f(0) (eq. 4.57) out to n terms.

$$\begin{aligned} f(0) =& P(M) + \frac{1}{2}pn\, P(B)P(\Omega) \\ &+ \frac{1}{2}pn\, P(B)P(M) + \left(\frac{1}{2}pn\right)^2 P(B)^2 P(\Omega) + \dots \\ &+ \left(\frac{1}{2}pn\right)^{n-1} P(B)^{n-1}P(m) + \left(\frac{1}{2}pn\right)^n P(B)^n P(\Omega) \end{aligned} \tag{4.58}$$

$$\sum_{i=1}^{\infty} \left(\frac{1}{2}pn\right)^{i-1} P(B)^{i-1}P(M) + \left(\frac{1}{2}pn\right)^i P(B)^i P(\Omega) \tag{4.59}$$

For solving the convergence of the geometric series we have $\sum_i^\infty a_i = \frac{a_1}{1-k}$, where $a_1 = P(M) + \frac{1}{2}pn\, P(B)P(\Omega)$, and $k = (\frac{1}{2}pn)P(B)$, which gives equation 4.60.

$$f(0) = \frac{P(M) + \frac{1}{2}pn\, P(B)P(\Omega)}{1 - (\frac{1}{2}pn)P(B)} \tag{4.60}$$

First of let us look at how this all works with the bank solution. As we already now the bank solution has two ways to define P(M), and P(B), one while $s_1 <= 0.5$, and another when $s_1 >= 0.5$. We will only be using $s_1 <= 0.5$ however, because we will put a upper bound on s_2 such that it is at max $s_2 = 2s_1$. Since $s_2 <= 1$, we cannot have $s_1 > 0.5$.

which gives us $P(M) = P(B) = \frac{s_1}{S(\Phi)+s_1}$.

$$f(0) = \frac{\frac{s_1}{1+s_1} + \frac{1}{2}pn\,\frac{s_1}{1+s_1}s_2}{1 - (\frac{1}{2}pn)\frac{s_1}{1+s_1}} \tag{4.61}$$

$$f(0) = \frac{s_1(2 + pns_2)}{2 + s_1(2 - pn)} \tag{4.62}$$

What we end up with is the $f(0)$ seen in equation 4.62. Let us now evaluate it with $pn = 1$ (eq. 4.63).

$$f(0) = \frac{s_1(2 + s_2)}{2 + s1} \tag{4.63}$$

From equation 4.63 we see the ratio $\frac{2+s_2}{2+s1}$, and that if $s_1 = s_2$, then we have $f(0) = P(H)$. But if $s_2 > s_1$ then $f(0) > P(H)$. This means that having different stakes in separate neighbourhoods is in fact a possible exploit. Looking back at equation 4.62 we find that decreasing pn will make it more difficult as pns_2 decreases while $s_1(2 - pn)$ increases. However we also have that the greater the ratio between $s1$ and s_2 is the greater the rewards are for the malicious actor. If we look at the total stake in a neighbourhood $S(\Phi)$, where it can be any positive number, the more stake committed to the neighbourhood or in other words the larger the neighbourhood, the easier it is to give yourself a small fraction of the stake. And the smaller the neighbourhood the easier it is for the storage node operator to give himself a larger portion of the stake. One takeaway is that the malicious operator is incentivized to join larger established neighbourhoods and act maliciously, while acting honest in newer smaller neighbourhoods, in order to maximize the stake ratio difference. With a stake difference of 10, i.e $s_2 = 10s_1$, this limits $s_1 <= 0.1$, but it will be profitable with $pn >\approx 0.2$. This makes the following exploit the most viable that we have looked at so far. Although it does appear to require a lot of micro-management, in order to position stake correctly in the correct neighbourhoods, and maintain that stake ratio. If done wrongly the exploiter can just as easily get less reward, than by playing honest in every neighbourhood, though that is mainly dictated by pn.

Let us take a look at how this works for the alpha solution, we now have that $P(M) = s_1^{\alpha+1}$, and $P(B) = 1 - s_1^{\alpha+1} - (1 - s_1)^{\alpha+1}$. We can start by substituting $P(M)$, $P(B)$, and $P(\Omega)$ into equation 4.60.

$$f(0) = \frac{s_1^{\alpha+1} + \frac{1}{2}pn\,(1 - s_1^{\alpha+1} - (1 - s_1)^{\alpha+1})s_2}{1 - \frac{1}{2}pn(1 - s_1^{\alpha+1} - (1 - s_1)^{\alpha+1})} \tag{4.64}$$

$$f(0) = \frac{2s_1^{\alpha+1} + pn\,(1 - s_1^{\alpha+1} - (1 - s_1)^{\alpha+1})s_2}{2 - pn(1 - s_1^{\alpha+1} - (1 - s_1)^{\alpha+1})} \tag{4.65}$$

Let us evaluate equation 4.65 with $\alpha = 1$. The alpha analysis we did before shows us that having $\alpha = 1$ is a good approximation for finding the turning point at which the malicious strategy overtakes the honest strategy, even for lower values of alpha. The main impact lowering alpha has is that it approaches the honest strategy more closely, which means that the expected losses decrease while below the aforementioned turning point. As well as the expected increase of rewards while s_1 is greater than the turning point.

$$f(0) = \frac{2s_1^2 + pn(1 - s_1^2 - (1 - s_1)^2)s_2}{2 - pn(1 - s_1^2 - (1 - s_1)^2)} \tag{4.66}$$

$$f(0) = \frac{s_1^2 - pns_1^2s_2 + pns_1s_2}{1 + pns_1^2 - pns_1} \tag{4.67}$$

Continuing on lets solve the equation $f(0) = P(H)$, for pn.

$$\frac{s_1^2 - pns_1^2s_2 + pns_1s_2}{1 + pns_1^2 - pns_1} = s_1 \tag{4.68}$$

$$s_1 - pns_1s_2 + pns_2 = 1 + pns_1^2s_1 - pns_1 \tag{4.69}$$

$$pn(-s_1^2 - s_1s_2 + s_1 + s_2) = 1 - s_1 \tag{4.70}$$

$$pn = \frac{1 - s_1}{-s_1^2 - s_1s_2 + s_1 + s_2} \tag{4.71}$$

Now let $s_2 = a\ s_1$, where $a >= 1$.

$$pn = \frac{1 - s_1}{-(a+1)s_1^2 + (a+1)s_1} \tag{4.72}$$

$$pn = \frac{1 - s_1}{(a+1)s_1(1 - s_1)} \tag{4.73}$$

$$pn = \frac{1}{(a+1)s_1} \tag{4.74}$$

If we want it solved for s_1 instead we just need swap the positions of s_1 and pn.

$$s_1 = \frac{1}{(a+1)pn} \tag{4.75}$$

From equation 4.74 if we have $s_2 = 2s_1$, or $a = 2$, then the maximum s_1 we can have is $s_1 = 0.5$, from which the minimum pn needed to profit is 0.66. While the minimum s_1 is $s_1 = 0.33$, however pn needs to be 1. If we compare this to the alpha solution analysis the equilibrium point has shifted down to $s_1 = 0.33$, from $s_1 = 0.5$, with $pn = 1$. Since $s_1 <= 0.5$, there is a $0.5 - 0.33 = 0.17$, s_1 stake window where having different stakes is a viable strategy. The entire window being one which requires $s_2 > 0.5$, in fact having the minimum $s_1 = 0.33$ in one half of neighbourhoods, and $s_2 = 0.66$ in the other is the same as having $s_x = 0.5$ in every neighbourhood. Except from here there is a greater window for increasing s_x, and also for increasing the possible rewards.

As such we can conclude that the alpha solution is not any more vulnerable to having different stakes in each neighbourhood, than having the same stake in each. Both being quite infeasible due to the necessary amount of total stake needed to invest into the strategy. Furthermore the different stake strategy requires one to remain an honest participant of Swarm in some subset of neighbourhoods, which means providing storage capacity in those neighbourhoods. The bank solution however, is vulnerable using the different stake in each neighbourhood approach. As opposed to not being vulnerable while the stake fraction remained the same in every neighbourhood.

4.5.3 Effect on other neighbourhoods

One concern about letting the pot carry over into new rounds is about how it affects the balance of neighbourhoods. In a malicious neighbourhood there are actors who reveal a different reserve commitment which gives the bank a chance to win. The storage incentives might select a malicious neighbourhood to earn the carried over pot, but most likely is that over time the reward will be given to neighbourhoods with zero, or close to zero stake committed to different reserve commitments, so called honest neighbourhoods.

Now this does sound like intended behaviour, we incentivize the nodes that are cooperating. The problem however, lies in how each neighbourhood is responsible to its assigned chunks. Ideally the number of nodes in each neighbourhood is uniform, such that each neighbourhood has as many nodes equal to the redundancy factor designated by Swarm, necessary to safely persist the chunk data from node churn. But also in order for the network to store as much data as possible. We postulate that few storage node operators would willingly join neighbourhoods where malicious behaviour has been registered, potentially leaving the data chunks of that neighbourhood at risk. On the other hand with a good data migration protocol to other (new) neighbourhoods, perhaps the network can combat malicious actors to some degree.

4.5.4 Burn

The simplest alternative to having the reward carry over into the next round, is to delete the reward, i.e burn it.

The immediate issue with analysing the impact of burning is that the storage incentives are no longer zero sum.

While it is not generally nice to burn money, one can say that all it does is deflate the bzz token. Which increases the value of the token for those who have already earned a reward. This is likely to have a similar effect on the neighbourhoods as what was discussed in section 4.5.3. Although less obvious, storage nodes in neighbourhoods without a chance for the bank to win, will be more attractive since they will earn bzz at a greater rate. While if the bank wins in any other neighbourhood, the value of their stockpiled bzz increases.

There is also a practical problem to consider, who is going to call the code to burn the reward? The current idea for reward distribution in the existing contract, is that once the redistribution game reaches the claim phase, whoever won the round will call the claim function. Although anyone can call the claim function, and the winner of the round gets the reward, if no one calls claim, the reward wont be handed out, and will be carried over to the next round. A safe assumption is that mainly the winner will be calling the function, because it costs gas and therefore tokens on the ethereum based blockchain where the smart contracts are hosted.

Essentially what this means is that one cannot rely on the storage nodes to burn the bzz, in the claim round, as whoever does it will lose ether. Thus there is a need for an additional bank smart contract, that calls the claim function in the case of the bank winning. This contract will need to foot the bill for calling claim to burn the money, and as such will be an expense from the Swarm team. One issue of this arrangement is that it is not a decentralized approach, as the network becomes dependent on a kind of central bank entity.

4.5.5 Bank it

The last option for what to do with the money when the bank wins, is to give it to the bank. After which it can be repurposed. First of this means that we once again need a bank smart contract to call claim, but this time it receives the reward. Which is going to offset the cost of calling claim. Secondly and more importantly are the implications of the Swarm team essentially taking a cut of the networks storage incentives. This is to an extent ethically unwise, as there now exists a conflict of interest between whoever is responsible for the smart contracts creating a safe redistribution game that minimizes malicious behaviour. And on the other hand allowing malicious behaviour to happen in order for the bank to win, which in turn rewards the smart contract hosts. To summarize we fear that allowing the bank to gain the reward makes the system vulnerable to corruption.

Chapter 5

Implementation

In this chapter we go through how we implement the discussed alpha, and bank approaches. And also touch on how we can test the correctness of the implementation.

We create two alternate redistribution.sol contracts, in the Solidity programming language version 0.8.1. And test them using the truffle test suite for Solidity.

5.1 the existing Redistribution.sol

During the following sections we are mainly going to be commenting on the parts of the original redistribution contract that we are changing in order to implement our solutions, my source for the original is this one: [24]. And as such we will in this section give a quick primer on how the original is implemented.

The way that the redistribution game operates was already discussed in section 4.2. So we know that there is a commit, reveal, and claim phase. Which phase the contract is in is decided by the current block number in the blockchain, and the round length. The round length being set as a number of blocks in the blockchain, and thus lasting for the duration it takes for the chain to mine this many blocks. The commit phase is the first quarter of the round, it is checked by taking the block number modulus the round length, and checking if that is less than a quarter times the round length. Similarly the reveal phase lasts for half the round length, and comes after the commit phase, subsequently the claim phase takes place in the final quarter of the round. The roundLength variable in this contract is set to 152, and the current round is the block number divided by the round length.

Each phase has a function that can only be run in that phase: commit, reveal, and claim. In addition the claim phase has the isWinner view, that has the same winner checking logic as the claim function, but does not hand out the reward. In solidity views do not cost gas when called externally. The claim function is intended to only be ran once by the winner after first having checked the isWinner view, since it is a gas expensive operation.

The commit function takes in an obfuscated hash, overlay, and the number of the round to commit in. It checks if its commit phase, and if the round number given is the current round. As well as if the overlay belongs to the caller of the function, if that overlay has at least staked the minimum stake, and that the overlay has not already committed in this round. The commit function simply takes the overlay and the obfuscated hash, and collects commits for the round.

The reveal function, takes overlay, and parameters used to create the obfuscated hash committed, the reserve-commitment hash, storage depth, and nonce. It checks if the overlay has committed and if the given parameters hash to the previously obfuscated hash. Crucially the reveal checks whether the overlay for the commit and the reported depth is within proximity order of the currentRoundAnchor as it is in code, or neighbourhood selection anchor as we understand it. And if every check passes, it collects a reveal for that overlay.

Finally the Claim function, which is the function that is subject to change with our solutions. It takes no parameters as all necessary data, commits and reveals, were collected in previous phases. It randomly selects a a truth from the reveals, and the final winner, using a corresponding anchor. Each anchor is decided on by a seed taken from the blockchain as seen in code snippet B.1.

```
        function updateRandomness() private {
        seed = keccak256(abi.encode(seed, block.prevrandao));
    }
```

Listing 5.1: Setting the seed for each selection anchor

Both the truth selection and winner selection parts of claim share the same selection mechanic. They will go through all reveals, generate a random number using the keccak256 hash of the truth selection anchor and the index of the current reveal in the reveal array. Then they add the stakeDensity of that reveal to the current sum. And evaluate winner based on inequality 5.1, where MaxH is the maximum value of the keccak hash. Which is coded as in inequality 5.2, to compensate for solidity's lack of floats. The way this works is that the first reveal will always win, but will be offset by the chance of winning that all the subsequent reveals in the array have. Each having a somewhat higher chance of winning than normal, until the last reveal, which has exactly as high chance of winning as its stake divided by total sum of stake in that round.

$$\frac{randomNumber}{MaxH + 1} < \frac{stakeDensity}{currentSum} \tag{5.1}$$

$$randomNumber * currentSum < stakeDensity * (MaxH + 1) \tag{5.2}$$

In the original contract every reveal not equal to the selected truth will be unable to participate in the winner selection, since its stake will be frozen. After the winner selection, the redistribution contract will tell the PostageContract to transfer the reward pot to the winner. And give feedback on how many correct participants there were in the winner selection to the PriceOracle contract.

5.2 Shared code

As previously stated in section 4.4.3, both versions use the **stake belonging to my reveal**, criterion. And therefore share the part of the implementation that keeps track of this metric. I build a mapping from the reveal hash (bytes32), to stake (uint256) (listing 5.2). Maps in solidity do not keep track of the keys, and that is why it is needed to keep an additional array to store the keys. The keys are needed primarily so that we can reset the map for every round, as simply using the delete keyword on a map is not inherently supported in solidity version 0.8.1.

```
bytes32[] currentRevealHashes;
mapping(bytes32 => uint256) currentRevealToStake;

function _resetRevealToStake() internal {
for (uint256 i = 0; i < currentRevealHashes.length; i++) {
    delete currentRevealToStake[currentRevealHashes[i]];
}
delete currentRevealHashes;
}
```

Listing 5.2: reveal hash to stake mapping

We have chosen to alter the reveal function to update the map for every submitted reveal. It is fitting as its purpose is already to handle incoming reveals, and it saves me an extra for-loop as opposed to if we where to do it in the claim function. In listing 5.3, we show how we update the mapping by taking the reserve commitment, (_hash function parameter), and adding the stake from the commit corresponding to the reveal.

```
//build reveal map
     currentRevealHashes.push(_hash);
     if (currentRevealToStake[_hash] == 0) {
          currentRevealToStake[_hash] = currentCommits[i].stake * uint256(2
               ** _depth);
      } else {
          currentRevealToStake[_hash] += currentCommits[i].stake * uint256
              (2 ** _depth);
      }
```

Listing 5.3: updating the reveal to stake map

We have for both versions also removed (commented out), any code that pertains to the freezing of node's stake. However, we did not remove the logic which selects the true reveal, which is ultimately an oversight that costs me a fair bit of gas. It does make it possible to retroactively freeze nodes, looking at the truthSelected event, and also easier to reimplement freezing if we want to. We will however, need to make use of this for loop later in each version anyway. Furthermore in both versions when no storage node has won (bank win). We let the reward pot carry over into the next round, by simply not making the Redistribution contract withdraw from the Postage-Contract.

5.3 Alpha

Equation 4.22 is how we will be selecting a winner with the alpha solution. We will generate a $RandomNumber$ and ensure its between 0 and 1 by dividing it with the $maxRandomNumber$. While the lefthandside represents a node's chance to win as introduced in section 4.4.4. If this chance to win is greater than the random number, then we select that node as the winner of the redistribution round. To implement equation 4.22 in solidity we have to do some tweaking. Because there is no support for floating point numbers, and as a consequence also no nth root function for α.

$$\frac{RandomNumber}{maxRandomNumber} < \frac{s_i}{S(\Phi)} * \frac{S(v(node_i))^{\alpha}}{S(\Phi)^{\alpha}} \tag{5.3}$$

The first thing we need to do for inequality 5.3, is to make sure there are no more division operations. By multiplying with every divisor on both the right hand side, and the left hand side, this can be accomplished. And the result is inequality 5.4.

$$RandomNumber * S(\Phi) * S(\Phi)^{\alpha} < s_i * S(v(node_i))^{\alpha} * maxRandomNumber \tag{5.4}$$

Now we have an inequality similar to the one used in the existing contract seen in inequality 5.2. The most natural thing to try is to substitute the old one with the new one. However using the same approach as before of going through each reveal and iteratively calculating the current sum of stake does not work. This is because the winning chances for each node no longer inherently sum up to 100 %. They only do so in the case there is only one unique reserveCommitment. What will happen if we keep the old approach, but change the criterion function? The first reveal gets a 100 % chance of winning, and the subsequent reveals decrease the first reveals chance of winning by a much smaller amount than before. This gives the first reveal processed a disproportionate chance of winning, and crucially the bank will never be able to win.

What we will have to do is calculate the total sum of stake beforehand. Giving everyone the correct proportional chance to win, to begin with. This must be done in a for loop before the winner selection for loop. Since we never removed the truth selection, we can use the sum calculated there. And with that we will also need to calculate the stake belonging to each reveal beforehand, which is a process I have already shown.

Using the same process as before but replacing the iterative currentSum, with a precalculated currentSum still has an issue. By creating a new random number for each reveal, then setting a new winner each time the criterion inequality evaluates to true, then the winner chances of each node are not independent of each other.

To create make every node's chance of winning independent, we have gone with a wheel of fortune themed solution. We had to reduce the number of random numbers from k to one, based only on the winner selection anchor. What we did was assign every node a disjoint slice equal to the right hand side of inequality 5.4.

In the code in listing 5.4, we have the version with $\alpha = 1$. It is part of a loop that loops through all the reveals in the round, obtaining a new revIndex each loop, and incrementing k. The first line is just sanitizing the randomNumber so it is not possible for it to be higher than MaxH. The second line is a bit hard to read as it spans three lines in this document, what it does is define the end of the wheel slice that this reveal has. The first slice will start at 0, that is the currentWheelSliceStart variable will be 0. And as previously mentioned we add on the right hand side of inequality 5.4, to the current wheel slice start. In order to create an interval: $[start : end]$, that the random number can land in, for reveal k to win proportionately to stake as in equation 4.22. On line 3 the left hand side of inequality 5.4 is calculated. We must then make two separate comparisons in the if sentence starting at line 4. One for if the random number calculation is greater or equal to the start of the interval, and another to check if it is less than end of that interval. If both of these apply we have a winner, and it is possible to break the loop early with a break; statement. The reason this is possible is because the slices for each reveal are disjoint, so if one of them wins, then there is no reason to check the others, because it would no longer be possible for them to win, using this wheel of fortune approach. Finally if the current reveal is not the winner, we set the start of the wheel slice equal to the end, and test the next reveal.

```
1          randomNumberTrunc = uint256(randomNumber & MaxH);
2          currentWheelSliceEnd =  currentWheelSliceStart + (currentReveals[
               revIndex].stakeDensity*currentRevealToStake[currentReveals[
               revIndex].hash]) * (uint256(MaxH) + 1));
3          uint randCalc = randomNumberTrunc*currentSum *currentSum * alpha;
4              //do initially with alpha ==1
5          if (
6              (currentWheelSliceStart <= randCalc) &&
7              randCalc <
8                  currentWheelSliceEnd
9          ) {
10             winner = currentReveals[revIndex];
11             break;
12          }
13         currentWheelSliceStart = currentWheelSliceEnd;
14         k++;
```

Listing 5.4: wheel of fortune winner selection

If a regular winner is not selected during the aforementioned loop. Then we say the bank wins. And currently that means that the reward for this round will be added to the reward pot for the next round.

A major challenge is setting the alpha tuning parameter. Currently we are implementing square root, and cube root, using the Babylonian method. One can see the square root function in listing 5.5, and the cube root function in listing 5.6.

Adding the α tuning is as simple as calling the root function on the part in the

```
        function sqrt(uint x) public pure returns (uint y) {
            if (x == 0) return 0;
            else if (x <= 3) return 1;

            uint z = (x + 1) / 2;
            y = x;
            while (z < y ) {
                y = z;
                z = (x / z + z) / 2;
            }
        }
```

Listing 5.5: Square root function

```
        function cbrt(uint x) public pure returns (uint y) {
            if (x == 0) return 0;

            uint z = (x + 1) / 3;
            y = x;
            while (z < y) {
                y = z;
                z = (x / (z * z) + 2 * z) / 3;
            }
        }
```

Listing 5.6: Cube root function

code that corresponds to the left hand side, or right hand side of inequality 5.4, respectively. As seen in listing 5.7.

```
      randomNumberTrunc = uint256(randomNumber & MaxH);
     currentWheelSliceEnd =  currentWheelSliceStart + (currentReveals[
         revIndex].stakeDensity*cbrt(currentRevealToStake[currentReveals[
         revIndex].hash]) * (uint256(MaxH) + 1));
    uint randCalc = randomNumberTrunc*currentSum *cbrt(currentSum) * alpha;
```

Listing 5.7: adding alpha tuning

5.4 Bank

We can now discuss how the bank solution is implemented. And as discussed earlier it is substantially easier to implement.

First of the bank needs an inevitable look through the reveal to stake mapping. Because to determine its own stake, it needs to find the reserveCommitment with the maximum amount of attached stake. In listing 5.8, we use the truth selection for loop, to look through the reveals, and find the reveal hash with the most stake from all reveals attached to it. Courtesy of the currentRevealToStake map, that we made before.

After finding the reveal commmitment with the most stake. Then we need a simple calculation to find out the stake of the bank (eq. 4.23). Implemented in listing

```
    // Find the maximum stake reveal
    if (currentMaxReveal < currentRevealToStake[currentRevealHashes[revIndex
        ]] ) {
        currentMaxReveal = currentRevealToStake[currentRevealHashes[revIndex
            ]];
    }
```
Listing 5.8: Looping through reveals to determine the Reveal with the most stake attached

5.9.

```
            uint256 bankStakeDensity = currentSum - currentMaxReveal;
```
Listing 5.9: calculate bank stake

Now we need to introduce the bank into the winner selection. Since it is possible to make use of the existing approach discussed earlier to implement the bank solution, that is what we are doing. In this regard what we could do to add the bank player into the game is to give the bank a mocked commit, with a corresponding reveal. But this muddies the arrays responsible for handling node commits, that in theory should hold legit data. What we have done instead is a slightly ugly yet verbose solution, of having the for-loop that goes through the commits go one extra step past the commits array length (k+1). And on this step handle checking if the bank will win, see listing 5.10. Now since the winner to be emitted in the solidity event is of a reveal type. We still

```
    if (i >= commitsArrayLength){
                randomNumber = keccak256(abi.encodePacked(
                    winnerSelectionAnchor, k+1));
                randomNumberTrunc = uint256(randomNumber & MaxH);

                currentWinnerSelectionSum += bankStakeDensity;
                 if(
                            randomNumberTrunc * currentWinnerSelectionSum <
                            bankStakeDensity* (uint256(MaxH) + 1)
                        ) {
                           winner = Reveal({
                         owner: address(this),
                         overlay: keccak256("BANK"),
                         stake: bankStakeDensity,
                         stakeDensity: bankStakeDensity,
                         hash: keccak256("bankhash"),
                         depth: 0
                     });

                        }

            }
```
Listing 5.10: Bank solution winner selection

have to create a reveal representation of the bank. One can see the details in the above code (listing 5.10), but the main takeaway is that the redistribution contract itself is the owner. And the overlay address of the bank is the keccak256 hash of BANK. Additionally the calculated bank stake is emitted. An important thing to note is that no storage node is likely to run the claim function in the event that the bank wins, due to the fees of the transaction not being counter balanced by a received reward. In reality the isWinner view will return the overlay of the bank, and in this case the storage nodes will not run claim, which justifies the need for a recognisable bank overlay. Furthermore, the bank reveal is still useful for testing the claim function.

5.5 Testing Results

Here we are testing the contracts on if they are able to reproduce the example written about in section 4.4.4. This is done to check if the contracts are implemented correctly, as in each provides reasonably corresponding results. In particular we are testing to see if we can correctly tune with the α parameter. And how much doing so will cost us in gas. It also allows us to confirm that the math done in section 4.4.4 is correct. However we will assume that an implementation error is more likely in this case, than an oversight in the mathematical model.

We simulated 1000 redistribution game rounds with different smart contracts using the truffle suits and collected data of which participant r_1-$bank$, was the winner. The results of the test can be seen in table 5.1. In table 5.1 each contract has two rows, the first being the expected results found earlier in table 4.4.4, the second the experimental test results. The difference column is the sum of the absolute difference between the expected values, and the experimental results. For comparison we also ran the existing Swarm redistribution contract to see what an acceptable difference would be. We refer to it as the baseline.

	r_1	r_2	r_3	bank	difference
$\frac{s_i}{S(\Phi)}$	0.500	0.400	0.100	tbd	
baseline win %	0.500	0.400	0.100	0	
(existing contract)	0.511	0.403	0.086	0	0.028
bank win %	0.333	0.267	0.067	0.333	
	0.348	0.269	0.060	0.323	0.034
$\alpha = 1$ win %	0.250	0.160	0.010	0.600	
	0.241	0.162	0.006	0.591	0.024
$\alpha = \frac{1}{2}$ win %	0.350	0.250	0.030	0.370	
	0.335	0.267	0.024	0.374	0.44
$\alpha = \frac{1}{3}$ win %	0.400	0.290	0.050	0.260	
	0.386	0.311	0.052	0.251	0.46

Table 5.1: Testing results

We are able to get corresponding results to the example in section 4.4.4. With a fairly low difference, less than 0.50 in all cases. $\alpha = \frac{1}{2}$, and $\alpha = \frac{1}{3}$, have a slightly greater difference than the others not using root functions, however. Which could be due to imprecision in the root functions.

α	Average Gas	Minimum Gas	Maximium Gas
Baseline	216925	207116	326357
BankContract	196467	154222	320069
1	169983	141129	239663
$\frac{1}{2}$	263696	265061	371173
$\frac{1}{3}$	473145	322403	757759

Table 5.2: Gas prices for implementation of different α

In table 5.2 one can see that the gas cost grows very rapidly when using root functions as a tuning parameter. It also shows that when α is equal to one, the performance is better than the bank solution, and the baseline. This is likely due to the wheel of fortune approach being able to skip out of the for loop that selects a winner early. The BankContract performs similarly to the baseline as one would expect since they are implemented almost the same, but has a significantly lower minimum gas cost. Which is primarily due to us removing the logic that checks and handles freezing.

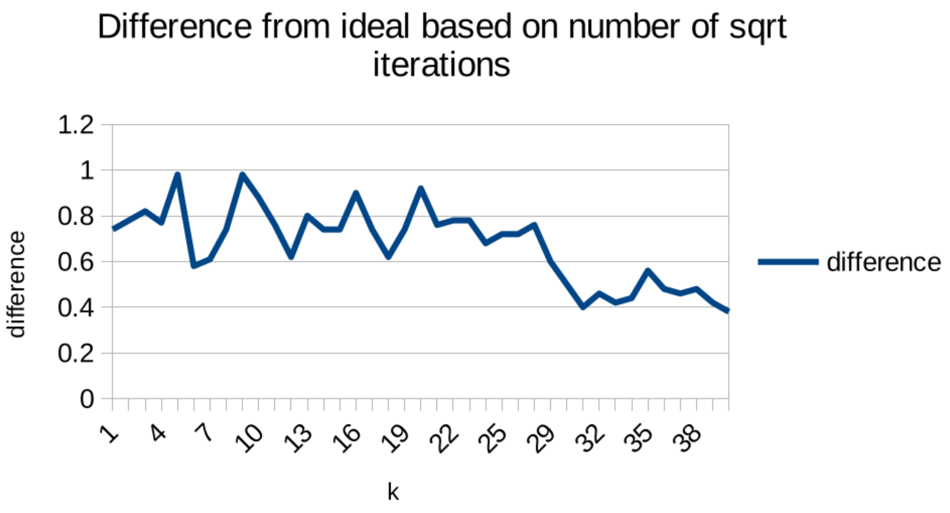
Can we optimise root functions?

The previous test showed that using root functions was a major gas sink. To investigate further if we can optimise the contracts gas cost, we added an additional constriction on the root functions: that they would run only a maximum number of iterations k. By limiting the number of iterations, we expect to observe a lack of accuracy from the functions

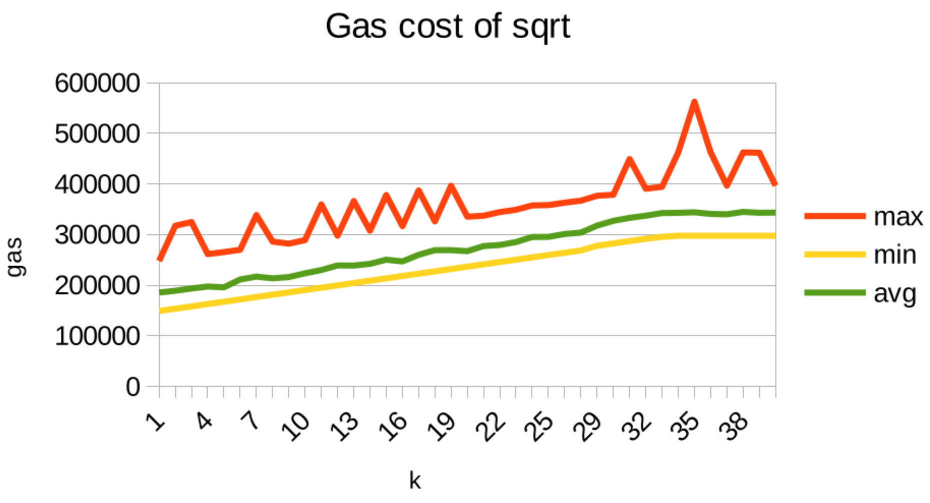
We implemented the constriction by adding a uint8 integer i variable to each function, and during the while loop checking if i was less than the maximum number of iterations k. And then incrementing i on every iteration. In essence this is a smart programming pattern to stop a while loop that might theoretically run an infinite number of times. The catch however is that it introduces another few instructions to run per while loop, further increasing the gas cost by its inclusion.

An additional thing to note is that we for this experiment did not break the winner selection loop when a winner was selected, letting the root functions run as many times as possible each round. In order to remove the variable of the loop ending early to the result, and to see the max cumulative gas cost reduction possible.

For each k we ran 100 rounds of the redistribution game, to produce the presented



(a) Sqrt performance



(b) Sqrt cost

Figure 5.1: Testing result for the square root function

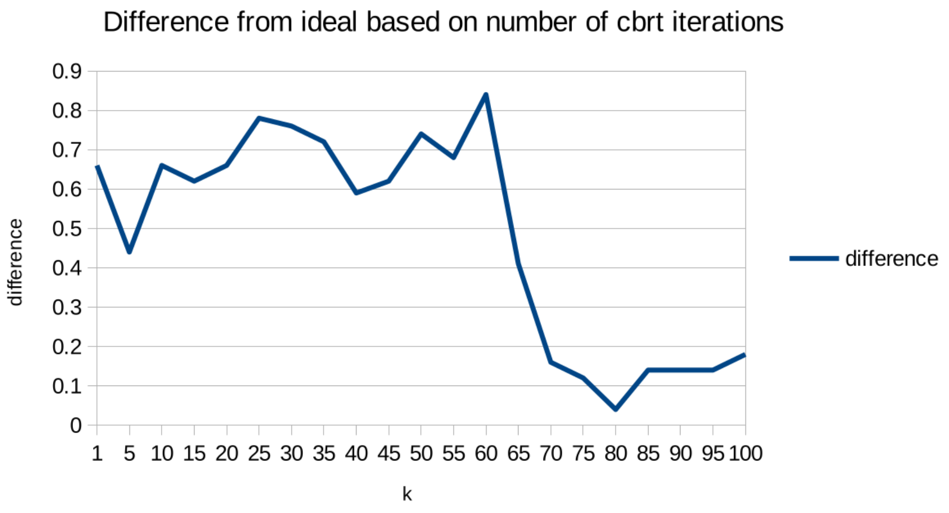
data. We took measure of the difference as before in 5.1, using the same example, and the minimum, maximum, and average gas costs of running the claim function. We are using difference as a performance metric, to see if the square root is able to produce correct results, even while being limited in iterations. The lower the difference the more correct results come from the functions, although we do expect some difference due to the stochastic nature of the redistribution game, especially at 100 rounds. In figure 5.1, the first plot has the difference plotted (5.1a), and the second the gas costs (5.1a). The x axis for both being the value of k.

What we found is that the minimum gas cost will stabilise at a value and remain there, as k grows, see the yellow line in plot 5.1b. The significance here is that this marks the point where k is no longer the limiting factor, when it comes to the number of iterations the while loop makes. And it is exactly here that the accuracy of the function peaks, if we look at the difference in plot 5.1b.

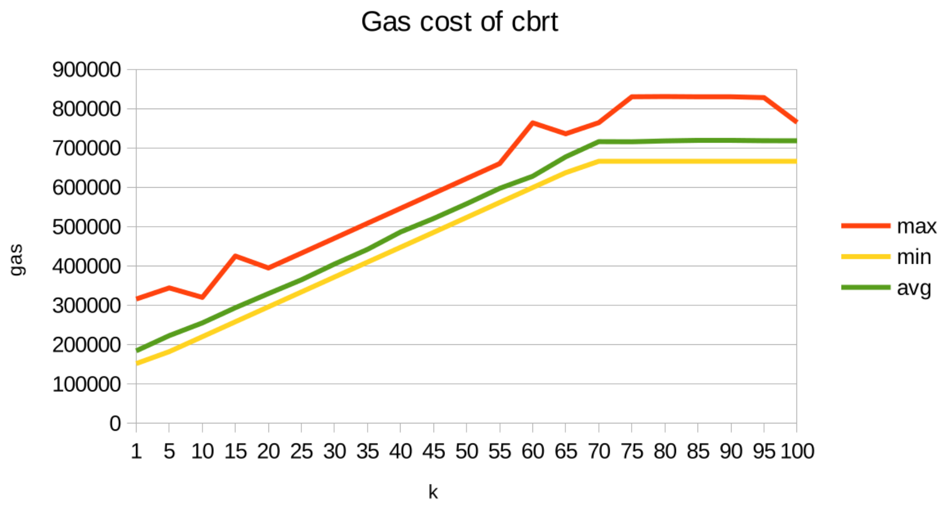
We ran the same test for the cube root function as well seen in figure 5.2, with one difference, we only tested every fifth k value, but we tested up until $k = 100$. The cube root function was expected to take more iterations, since it was already known to cost more gas than average than the square root function. And to save considerable time we increased the interval between each k.

The results from testing cbrt (fig. 5.2), show the same thing that we pointed out for squareroot, and even more clearly. When increasing k, no longer matters, as in the gas cost stops growing, it is the exact time the performance improves.

From this we can conclude that the root functions are already optimised. At least as far as we can tell. Which unfortunately means that there is no gas reduction possible by altering these functions. And while enforcing a limit k might be good sanity wise, such that we know each while loop will at some point end. It is an increase in gas whose cost outweighs the gain, and as such we will not be using it outside of this test.



(a) Cbrt performance



(b) Cbrt cost

Figure 5.2: Testing result for the cube root function

Chapter 6

Experimental Evaluation

In this chapter we will be discussing how to evaluate the effectiveness of the new storage incentive proposed in chapter 4. And show the experimental results of our evaluation.

6.1 Experimental Setup and Data Set

Our purpose is to use real data of previous redistribution rounds, in order to see how those rounds would have turned out had they used our Redistribution contract. And with that evaluate the effectiveness of the bank and alpha solutions. As while the implementation test in section 5.5

We are using data obtained from the swarmscan API [30] spanning roughly the year of 2023. We are collecting data from rounds numbered: 205956, to 180241. These rounds happened originally between the 6Th of December 2023, and the 22 of December 2022. Rounds older than 180241 do not provide me with the necessary selection anchors, and as such we cannot use them. This is likely due the smart contract used by Swarm being updated at this time, though it could also be due to an API change. Rounds newer than 205956 use a newer smart contract developed by Swarm, than the one we are basing our work off of. The new contract by Swarm focuses on improving inclusion proofs, such that it is harder to submit a wrong reserve commitment. Although the round data from swarmscan is still in usable format for our testing beyond round 205956, the testing setup we will use makes it cumbersome due to time constraints to run a number of rounds greater than around 10000 rounds. As such we are not generally in need of more data.

The testing setup uses truffle to deploy our redistribution smart contracts to a local ganache blockchain. The pipeline for running each round is as follows. First we go through each reveal in the round and configure the blockchain accounts given to us by ganache. Each account is given bzz to match the stake in the reveal, and we precompute the obfuscated hash needed to be commited in the commit phase. If there are more reveals than we have blockchain accounts, then we loop around and use the same accounts again. We then advance the blockchain such that we are at the beginning of the commit phase, and start committing the obfuscated hashes calculated before. After which we run reveal, and finally claim, before logging the results of our simulation in a CSV file.

In order to accomplish this we need the following data from each round in order to run the round again for a new contract. Most importantly we need the reveals for that round, as this is where all the data that the nodes send in, is as the name suggests, revealed. The data contained in reveal is the stake of the node, the reserve commitment, depth, and overlay. With this data it is possible to create the obfuscated hash used to commit in the commit round. We require the neighbourhood selection anchor, to be able to pass a check in the redistribution contract. The test checks whether the reported depth and overlay in the reveal is within PO of the anchor. Or in more plain English: it tests that the node is a member of the neighbourhood that it claims to be in.

We feed the data from the API into a sqlite database for local storage. Having the data locally saves us from using time and network resources on retrieving the data from the API every time. Furthermore it lets us reliably be able to run tests and analyss The decision to use a SQL database for local storage was mainly motivated by the ability to issue powerful queries on the data to filter out, analyse, and retrieve the necessary data for testing.

The data used in the test is gained by getting all the round numbers of frozen rounds from the database. A frozen round is a round where there was originally frozen nodes in the neighbourhood at either the onset of a round, or after it. Meaning that the round generally had different reserve commitments revealed. By doing this we can test our solutions on every potentially problematic round.

The pie chart in figure 6.1 shows how the rounds retrieved from the API are distributed. There are four categories: honest rounds, rounds with majority reveal, chaotic rounds, and frozen in previous round. Of which the rounds with majority reveal, chaotic rounds, and frozen in previous round, are subcategories of the 7580 frozen rounds found in our data. In total these frozen rounds make up for 28% of all the rounds. Rounds with majority reveal, mean that we have 50% or greater of the reveals voting for, or including the same reserve commitment. While chaotic rounds means we have less than 50% of reveals with the same reserve commitments. These categories account for 6611 rounds, and 584 rounds respectively. There are 385 rounds that are noted as frozen in previous rounds. What this means is that they possess no conflicting reveals due to all the disagreeing nodes being frozen in a previous round. In our case this means that they are unusable as test data, but serve as evidence of the issue that a faulty reveal can freeze the rest of the neighbourhood, to increase their reward in subsequent rounds. Of course most of these 385 rounds, most likely had the correct reveal chosen as truth. To clarify the remaining frozen rounds could also have frozen nodes in previous rounds, but, they in that case have new conflicting reveals in this round. There are 18135 honest rounds, we do not test these rounds since due to the way alpha and bank behave, these would not perform

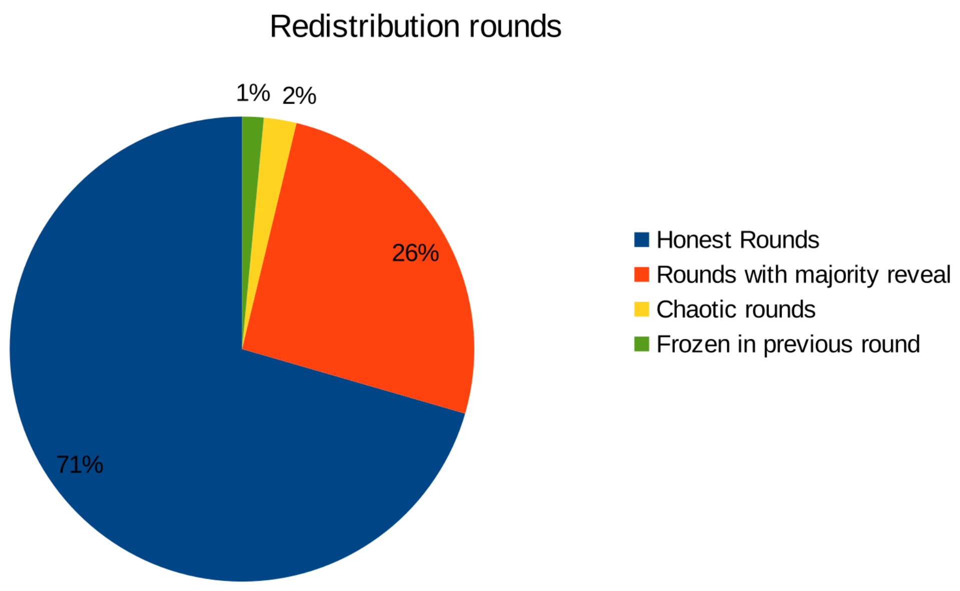


Figure 6.1: Pie chart of retrieved data

any differently than the original redistribution contract. In the bank solution, the bank player would not get any stake, and in the alpha solution the factor considering stake belonging to reveal, would evaluate to 1.

This leaves us with 6611 rounds with majority reveal, and 584 chaotic rounds that we can use for testing. In the following section I will present the result of each of these two categories separately.

The settings used for the ganache network are as follows. We set the block timing, to automine, which means for every transaction, the blockchain advances one block ahead. This is the fastest setting for commiting transactions to the chain, and to be able to read events. We create 12 blockchain accounts, each with 100000000000000000 ether. Additionally we set the transaction gas limit to 8000000000000000, and the gas price to 1. These settings are in order to allow as many redistribution rounds as possible to be able to run. Each account has a more or less inextinguishable supply of ether to pay a minimum gas price for running transactions. While the gas limit is set high such that any transaction is allowed to complete.

The experiment took some time to run through. While hardware dependant we were able to run one set of 6611 rounds in around 5 hours using AMD Ryzen 5 5600x 12 cores cpu, with 32 GiB of DDR4 memory. As we ended up running 3 sets of repetition per storage contract, this took approximately $3 * 3 * 5h = 45h$, And this is only counting the useful runs. Running the chaotic rounds with 584 rounds went a lot smoother, as it did not need to risk losing connection to the ganache blockchain, or running out of memory. In this case each repetition took around $10 - 20$ minutes.

Before moving on to the results let us comment on the pros and cons of the testing setup. As stated before we are using the truffle test suite along with a local ganache blockchain. The reason why we are using truffle is since I already used it to test the smart contract implementation, the pipeline to run redistribution rounds is something we have already set up in truffle. In addition it is heavily integrated with ganache, which is the most convenient blockchain as far as we know to use to run the contract. And instead of looking for, or developing a better solution, it could just as well be better in this case to keep it simple, as in if I start testing in this way while thinking about other solutions, We will probably be done running the simulation of rounds anyway before figuring out a better way to do it. Swarm-SI or other simulation without a live blockchain is an option, but we wanted to keep it a realistic blockchain setting.

Truffle however is not the ideal way to run 7195 redistribution rounds, and the many smart contract function calls needed each round, as it is designed to be used for unit testing. Furthermore it emphasises setting gas limits and making sure the tests ran are manageable in gas such that the tested smart contracts are suitable to be deployed on a real ethereum blockchain. Even more restricting for us is that the tests have a set timeout, as in if the tests run for longer than this time it will timeout before completing. One can increase this timeout, but it was found that truffle will lose connection to the ganache server, and end the test after one hour when running the truffle test command with default settings. Which is why we needed to run the ganache network manually.

Even with the lenient ganache configuration, we still ran into issues where truffle could not connect to the blockchain after a long time had passed. Our current hypothesis as to why is because the blockchain had gotten too large and strained the computer system resources, making it less responsive. Restarting truffle with the same ganache instance, would make the connection to the network timeout sooner than for a fresh ganache blockchain. This meant we had to reinitalise the ganache blockchain every now and then, which made getting the test results a semi-automatic procedure. The most important setting to increase the speed of the simulation, is how often the ganache blockchain mines new blocks. In this regard we found that anything other than automine is dreadfully slow. We tried to run with the minimum block mining time of 1 second, but it was still much slower. Automine however, gives us a transaction limit to work with. Because the redistribution rounds, round length and the lengths of each phase, are decided by the block number on the blockchain. It can only process so many commit / reveal transactions before it rolls over into a new round, this worked for all but one redistribution round in the chaotic rounds where there were 45 reveals. In all likelihood the easiest fix to allow any round to be completed this way, is to increase the length of each round on the redistribution smart contract.

6.2 Experimental Results

6.2.1 Simulating with swarmscan data

We present the results of the swarmscan data testing in two batches, one for the reasonably well behaving rounds where one reserve commit is chosen by at least half of the storage nodes. And another batch for chaotic rounds where we cannot tell for the most part which reserve commitment is the real truth.

The x axis on figures 6.2-6.7 shows the fraction of stake that the reserve commitment with the most stake has in each round. When this fraction is denoted as 0, it means stake fractions $0 < s < 0.1$, as the highest stake in each round has to be nonzero. While the y axis is the rewards in each bin normalized to be between 0 and 1, and as such I have denoted it as the reward percentage for each fraction. The reason for normalizing is because each bin, each fraction does not have a uniform number of rounds in the data set. And as such they are not easily comparable in a raw format. Furthermore this means certain fractions have greater variability than others due to the lower number of rounds. The number of rounds for each fraction are also displayed on the plots. In particular the 0.4 fraction in figures 6.2, 6.3, and 6.4, only have 23 rounds worth of data. For the rounds with majority reveal, each plot is an average of 3 repetitions worth of data, with accompanying error bars. And for the chaotic rounds each plot is an average of 10 repetitions worth of data, but as the number of rounds for each fraction is generally much lower, the variability remains much higher.

The majority reveal for each round in context of the result plots, was decided by which reserve commit had the most stake attached. If the winner of a round had the majority reserve commitment its reward is added to the majority reward column (green). If not it is added to the minority reveal column (blue). That is unless the overlay submitted to the reveal matches the overlay given to the bank which is the keccack256 hash of ”BANK”. In which case the reward is denoted bank reward (red). The results presented are with the pot of the original rounds, which means it does not have the pot carried over to the next round. The reason not to do this is due to us having chosen specifically rounds with different reveals, and this is an issue because in many cases in reality, the carried over gold would go to an honest neighbourhood selected in the next round. On the other hand using reward carryover, could display a more worst case scenario, but it does make the results harder to read, since the bank reward carryover, would arbitrarily flow over into majority, or minority reward, in any of the stake fraction columns.

Rounds with majority reveal

The graph in figure 6.2 shows the result of running the rounds with majority reveal for the bank solution. We can see how both the majority reveal, and minority reveal get less overall expected reward due to the bank player. Looking at the graph the bank reward and minority reward are just about equal for every stake fraction. Fur-

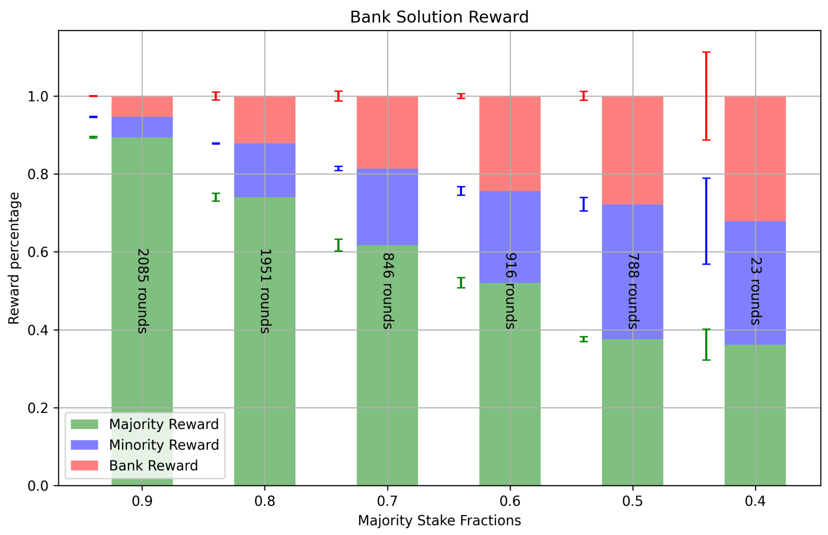


Figure 6.2: Result for bank solution

thermore we have when the stake fraction is greater than 0.5 that the majority stake fractions reward percentage is reduced relatively less than for the minority, which is overall positive.

The graph in figure 6.3 shows the result of running the rounds with majority reveal for the alpha s solution with $\alpha = \frac{1}{2}$. Here we see that generally the bank, and the minority factions get different rewards. With the bank winning much more often than the minority, compared to the bank solution, this is due to the alpha scaling discussed in sections 4.4.4, and 4.4.4. The consequences of the bank winning more often is mainly that the minority reveal is punished more harshly, Although each faction or strategy do appear to be equal when the stake fraction is 0.5. We have found before that stake fraction 0.5 is a turning point, and that any stake above that means the majority gets more reward (with bank reward carryover).

The graph in figure 6.2 shows the result of running the rounds with majority reveal for the alpha s solution with $\alpha = \frac{1}{3}$. Since α is lower this time the bank reward in this plot is less than it was for figure 6.3. Stake fraction 0.5 still marks a turning point where bank reward plus minority reward is now greater than the majority reward.

Chaotic rounds

We have for the chaotic round figures 6.5 , 6.6, and 6.7, that the most relevant bars are for the stake fractions between 0.5 to 0.1. Firstly because these are stake fractions we have yet to look at for the rounds with majority reveal, and secondly because these are the bars where we have the most rounds of data. One could say that the majority versus minority comparison is less relevant, due to it being harder to tell what the majority is in these rounds, and if that majority is submitting the correct reserve com-

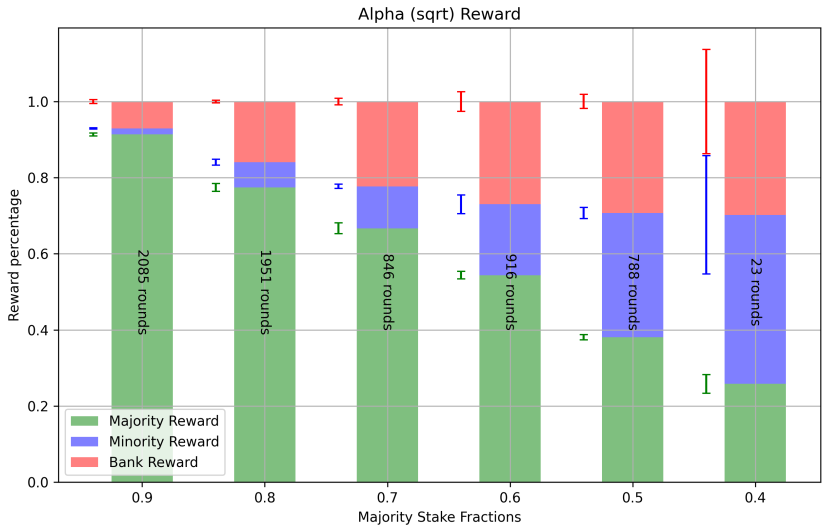


Figure 6.3: Result for alpha with $\alpha = \frac{1}{2}$

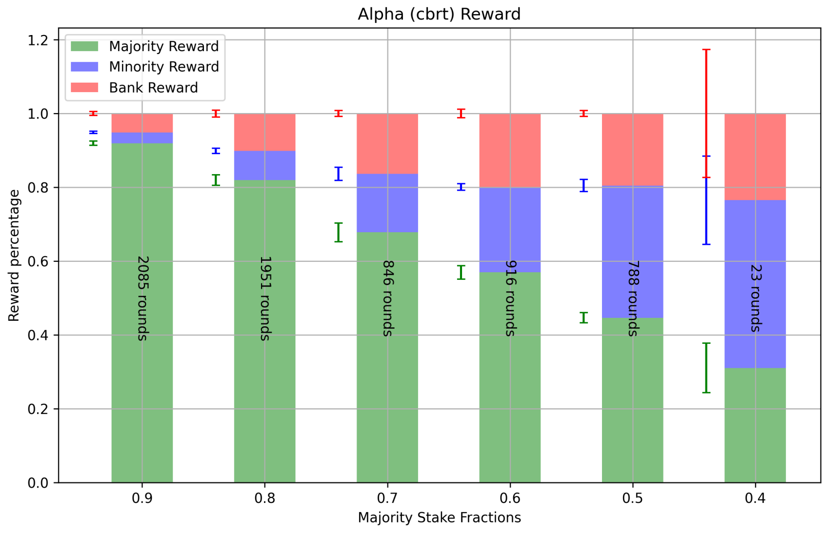


Figure 6.4: Result for alpha with $\alpha = \frac{1}{3}$

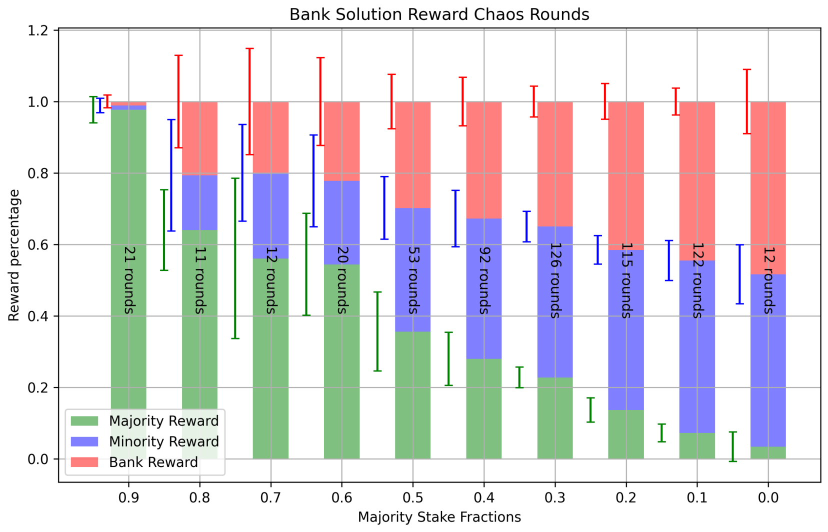


Figure 6.5: Result chaotic rounds for bank solution

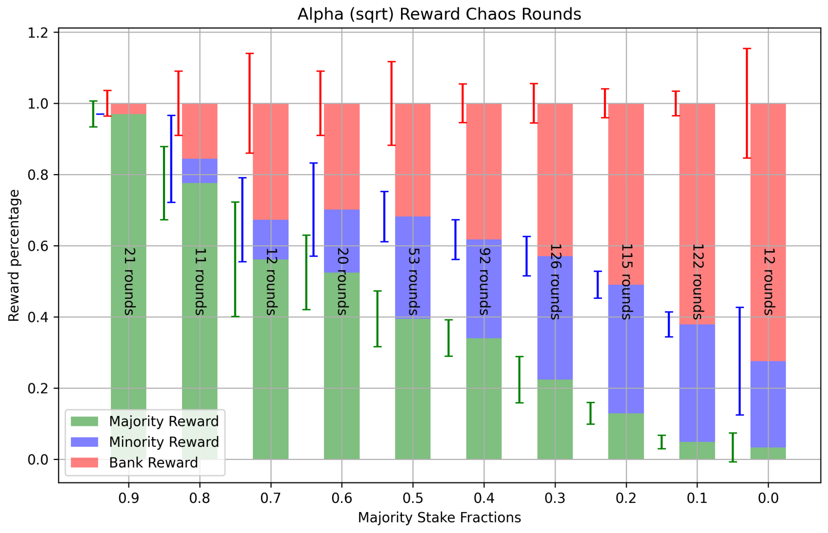


Figure 6.6: Result chaotic rounds for alpha with $\alpha = \frac{1}{2}$

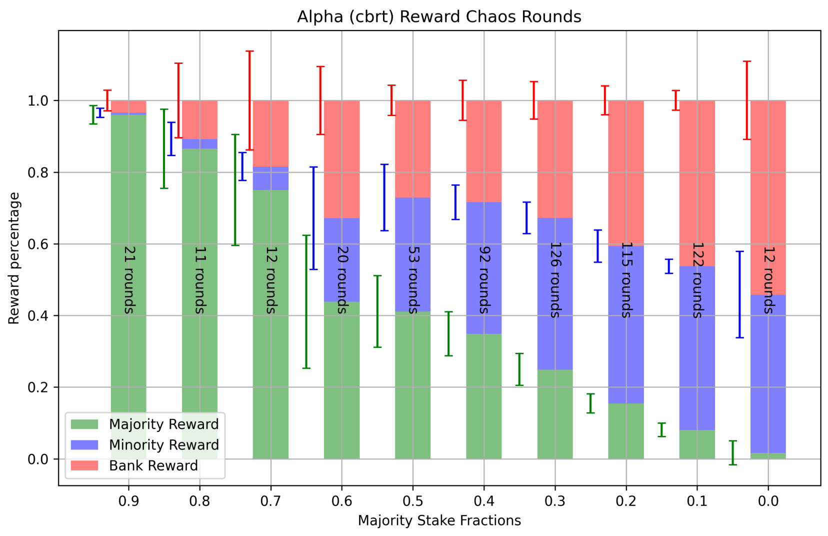


Figure 6.7: Result chaotic rounds for alpha with $\alpha = \frac{1}{3}$

mitment for the neighbourhood. However, the majority faction is still representing the reserve commitment with the most stake and is likely a result of multiple reveals voting for it. Even so as the stake fraction goes down, the likelihood of the majority just being a singular reveal from a node with slightly more stake than the others increases. Id like to point out again that the 0.0 fraction in the figures is representing stake fractions $0.1 > s > 0$.

The comments made for the rounds with majority reveal are all still applicable to the chaotic rounds. The alpha solution when $\alpha = \frac{1}{2}$ gives the bank more rewards than the minority (fig. 6.6), while the bank solution generally has around equal bank reward percentage, as minority reward percentage (fig 6.5). While when $\alpha = \frac{1}{3}$ the bank reward fraction is less (fig. 6.7), compared to when $\alpha = \frac{1}{2}$ (fig. 6.6). The main takeaway for the bank reward reward fraction to be so high overall is that we do not have much coordination among nodes, as in few nodes reveal the same reserve commitment.

Chapter 7

Discussion

In this chapter we will discuss our new redistribution contract solutions, and how they ultimately fare compared to the original approach.

The existing redistribution smart contract has neighbourhood nodes agree on which data chunks are being stored in said neighbourhood by calculating a reserve commitment hash. Each neighbourhood node submitting a hash has a chance to be selected as the true reserve commitment proportional to their stake, and subsequently the more nodes submit the same hash, the more likely that hash has to become the truth. What we see happening however, is that in many rounds there are many hashes different from the majority, or most likely truth, being committed, often times each different hash only being submitted by a singular storage node. The reason for these different hashes could be due to arbitrary misconfiguration or network error, but it is believed that a significant fraction of them are due to storage node operators looking to take part in the storage incentives, without doing actual storage. To mitigate such malicious free riding storage nodes, the existing contract has a freezing system which would by freezing nodes that did not commit the same hash as the selected truth, make them ineligible to take part in upcoming rounds until unfrozen. It would also make them eligible for being further punished by having their stakes slashed.

Our analysis of the freezing mechanism found that it actually facilitated the reveal of malicious reserve commitments. As in if one got lucky and had their reserve commitment chosen, it meant that everyone else is frozen, which means that when the neighbourhood is once again selected, the malicious storage node would be the only one to be rewarded. Ultimately we found this strategy to be equally profitable to being an honest storage node, but since this method allows free riding on storage it is of course a problem.

Our way of stopping malicious actors are two alternative coordination rewarding smart contracts. Both of which provide a chance for the storage nodes in a round to lose the reward entirely, when this happens we say the bank wins.

The first smart contract presented is the one we refer to as the bank solution, named after the mocked storage node (bank) player we create. This contract is implemented in much the same way as the existing contract, except for the definition of the stake that the bank player is playing with. Which is set to be relative to the total stake in the neighbourhood and the reserve commitment with the most stake attached.

The other smart contact alpha, gives each storage node a chance to win relative to its stake and the stake committed to that storage node's reserve commitment. We were unable to implement alpha in the same way as the existing contract due to the summed up chances for each player in the redistribution game to win is not equal to one. And as such the old strategy of giving the first player a 100 % chance of winning, offset by each subsequent player's chance of winning. Will not work since the offset is much smaller this time around with the alpha equation 4.22. It works for the bank solution due to the bank acting as a player. Instead we opted to use a wheel of fortune approach, which does perform a little better than the existing contract baseline with $\alpha = 1$. However, it only uses one random number instead of one for each player. The consequences for which we are unsure of, since it is our belief that if a player is able to obtain the random seed, to discover one of the random numbers, they can also find all of the other random numbers. Setting and using α is the main performance bottleneck, as square root and cube root functions cost us a lot of gas. By the testing done in section 5.5, where we looked at if it was possible to limit the number of iterations of these functions, we found no further way of optimising them.

The gas costs of both alpha and bank do increase over the original, due to the necessity of keeping track of the stake associated with each reserve commitment. However, great gas savings are made by removing the freezing mechanisms of the old contract, resulting in both the bank and alpha with $\alpha = 1$ having increased performance over the original in table 5.1, in section 4.4.4. Reintroducing freezing would ultimately make them perform slightly worse than the original. Alpha with $\alpha = 1$ could perform slightly better than the original depending on how much performance was gained by using the wheel of fortune implementation. Meanwhile the performance when $\alpha = \frac{1}{2}$, i.e when using square root performs slightly worse than the original contract. And it will perform even worse with the freezing mechanisms enabled. Which also applies to when $\alpha = \frac{1}{3}$.

Deploying the alpha contract with $\alpha = \frac{1}{3}$, is as it stands not viable due to the high gas cost, having a maximum recorded gas of 757759 in our testing (tab. 5.1). That is more than double the maximum gas of the original contract, which was 326357 gas in our testing. Looking at the maximum gas values allows us to make decisions based on the worst case scenario.

Continuing on when it comes to the alpha solution having α set to 1 is also not viable, due to the disparity between how larger stake fractions, and smaller stake fractions are punished. We discussed this aspect of α during the presentation of the alpha solution in section 4.4.4, and when discussing the implementation test in section 4.4.4. As it stands the only alpha smart contract that looks to be viable to deploy is when $\alpha = \frac{1}{2}$.

Both alpha and bank lowers the expected reward of a storage node submitting the wrong reserve commitment by providing the bank a chance to win. This behaviour can be observed by looking at the plots in chapter 6, for example in figure 6.2 containing results for the bank solution. In the 0.9 stake fraction column, the $0.9 - 1.0$ reward percentage interval would with the existing contract be all reward awarded to the minority, but with the bank contract it split with the bank.

The properties of the alpha function, and the bank solution differ slightly. The difference is rooted in how the different reward function for each solution look. We have $R(node_i) = \frac{s_i}{S(\Phi)+s_{bank}}$ (eq. 4.24) for the bank solution, and $R(node_i) = \frac{s_i}{S(\Phi)} \frac{S(v(node_i))^\alpha}{S(\Phi)^\alpha}$ (4.22) for the alpha solution. $S(\Phi)$ being the stake of the neighbourhood summed up, s_i stake of $node_i$, $S(v(node_i))$ is the stake belonging to the reveal of $node_i$. With S_{bank} referring to the bank stake defined as $s_{bank} = sum(S(v)) - max(S(v))$. As such for the bank solution its chance of winning depends on the reserve commitment with the most stake attached only, while the alpha solution is relative to how many other storage nodes also reveal the same reveal as $node_i$. This means that the alpha solution is more sensitive to other nodes working together. Since stake inherently provides some sybil resistance, as in an operator within a neighbourhood expects either the same or less reward, when splitting a node with x stake into nodes n with x/n stake. Then the alpha solution is mainly going to help nodes that aren't malicious, but which may come up with different reserve commitments for other reasons. Such as those storage nodes using different versions, or storage nodes being unable to obtain the chunks they need to store either by those chunks being withheld, or by network error.

By deploying the smart contracts on ganache with truffle, and simulating, we were able to experimentally evaluate the different contracts. In figure 6.2 where the bank contract was deployed compared to 6.3 with the alpha-contract ($\alpha 0\frac{1}{2}$), the bank is winning a greater amount with the alpha solution. In the same plots but for the chaos dataset (fig. 6.5 and fig. 6.6), this is much more pronounced. Ultimately the takeaway is that the nodes with minority reveal did not coordinate their actions as in each one reveal different reserve commitment hashes. And this behaviour is punished a lot more with the alpha approach.

We had some discussion about what to do with the reward if the bank wins. The one that comes out on top for the time being is to carry that reward over into the next round. The reason why being mainly that there is a practical issue with burning the reward, as to who pays the blockchain fees to do so, and that letting the Swarm team take that responsibility might not be aligned with the vision of a self regulating decentralised storage network. Giving the reward to the bank entity instead, does not have the fee paying problem however, it does produce a conflict of interest. That if the bank funds are managed by the Swarm team, they might want to at some point facilitate a high chance for the bank to win, in order to gain more funds.

Furthermore both burning, and carrying over the bank rewards, have the same problem. In that the neighbourhood which produces a chance for the bank to win, are inherently undesirable, since whoever joins such a neighbourhood will have a lower chance of winning, than if they would join another neighbourhood with all honest nodes. With a carried over reward, those honest neighbourhoods are likely to receive that carried over reward, making them more desirable. Likewise with burned bank rewards, the honest neighbourhoods are generally more likely to win, while the burned tokens create deflation increasing the value of tokens for those holding on to them. Arguably the effects of deflation in this regard are not as pronounced as when the reward is carried over however.

The reward being carried over introduces the storage incentives to some new possible exploits however. These are mainly possible with operators having nodes present in multiple neighbourhoods, to exploit that the additional reward might be a part of the pot for one of these nodes. We looked at two cases to game the system with nodes in multiple neighbourhoods, one with a node with the same stake in each neighbourhood. And another where the operator has two different stakes one being played in one half of the neighbourhoods the operator is part of, and the other stake value in the other half. We will refer to the first as 1), and the second as 2). For 1) we found that the bank solution is overall safe for this exploit, as the exploiter would need to be part of every neighbourhood. While the alpha solution has combinations of node stake, and fraction of neighbourhoods entered, that would allow the malicious actor to get more rewards than if acting honestly. These combinations however, still need so much stake that the exploit is thought to be possible in theory only.

In the case of 2), the bank solution has issues however as with a high enough difference between the two stakes, it is possible to accomplish this exploit, with a lower fraction of neighbourhoods entered. On the other hand the alpha solution is just about as resistant to the different stakes approach, if not more due to a decreased stake window where the exploit is possible.

In conclusion we have the more theoretically sound alpha solution, with better coordination rewarding, that is generally more resilient to exploits. If we do not consider the possibility of an actor or collective of storage nodes, managing to obtain a monopoly (51%) on neighbourhood stake. And the easier to implement, less expensive bank solution. The bank solution providing more or less comparatively fair results to the alpha solution, while having a lower gas overhead. Makes it the more viable solution for practical deployment. This can change if we manage to find a cheaper way to implement a contract with the same properties as the alpha solution.

One aspect of our analysis that might not translate well into the real world, is that we think of rewards obtained later as being as valuable, as rewards earned today. The affected parts are mainly our analysis of the existing contract, and the analysis of the bank carryover reward. Even so we can see in reality from swarmscan that redistribution rounds with multiple different reserve commitments continue to be a problem. Meaning there are many who aim for the possibility to be rewarded in later rounds, with the other neighbourhoods frozen. Additionally as we briefly discussed, when considering our approach for the solution, the storage incentives in Swarm are set up as an indeterminate game of chance where a single node takes all reward for a round. Which means that while the expected reward in the long term might be reliable income, in the short term it is unreliable.

With simulating the redistribution rounds using data from swarmscan we were attempting to show that the solutions are good enough to be used in practice. What we did not do however, is deploy the implemented contracts on real blockchains or more realistic test-nets, and by spinning up storage nodes with bee clients. While something one should do extensively before actual deployment, it is not as relevant to this thesis. The main advantage gained by our approach is the time gained by not having to wait for the next redistribution round, and the time gained in setting up and managing Swarm nodes.

What we have left to explore is how our new solutions would work while keeping the existing freezing mechanisms. Mainly how this could affect our mathematical analysis, that is to say would the weak Nash equilibrium for honest nodes in section 4.3.2 become a strong equilibrium ? And how would it affect the possibility of exploiting the carried over reward.

Chapter 8

Conclusions

In this thesis we have evaluated and redesigned the Swarm redistribution contract. The analysis of the existing contract revealed that a node revealing a different reserve commitment than other nodes is an equal strategy to cooperating on revealing the same reserve commitment. This due to the freezing mechanisms allowing the node with the different reserve commitment to freeze all other nodes in that neighbourhood. If the same neighbourhood is once again selected, then the malicious node is the only one eligible to win. Our approach to solving the issue is for the new contract to have a greater coordination rewarding incentive. We created two alternative solutions for the new contract, the alpha solution, and the bank solution. The alpha solution selects a winner proportional to a node's own stake, and the stake of nodes also submitting that node's reserve commitment. Whereas the bank solution adds another player in the game with stake based on the reserve commitment with the highest amount of stake in that round. Implementing the bank solution is easier, than the alpha solution. Through testing our implementation we found the gas costs of the alpha solution to be much higher than the bank solution. Although this is mainly due to the usage of root functions. This ultimately makes the bank solution best fit for practical purposes. However the alpha solution, does intuitively allow nodes not meaning to misbehave, but who produce different reserve commitments due to bee version mismatches. The ability to coordinate amongst themselves on the correct reserve commitment for that version. Furthermore the bank solution with the reward carrying over into the next round, is vulnerable to an exploit where the storage node operator has nodes in neighbourhoods with different stake amounts. While the alpha solution is resilient to this exploit, but does otherwise encounter problems if any one entity manages to hold the majority of stake in a neighbourhood.

Appendix A

Code repository

The code for this project is available at the following GitHub repository:

https://github.com/DanielHavstadUIS/MasterThesisSSC

This repository contains the implementations of the alpha and bank solution in solidity. Along with our testing setup using data from swarmscan. And data analysis of the results of simulating previous redistribution rounds with our new solutions.

The repository below contains modifications i have made to Kristian Tjessem's SwarmSI [16], while working on this thesis:

https://github.com/DanielHavstadUIS/SwarmSI-Sim

Appendix B

Bank solution contract code

```
      // SPDX-License-Identifier: BSD-3-Clause
pragma solidity ^0.8.1;
import "./@openzeppelin/contracts/access/AccessControl.sol";
import "./@openzeppelin/contracts/security/Pausable.sol";
import "./PostageStamp.sol";
import "./PriceOracle.sol";
import "./Staking.sol";

/**
 * @title Redistribution contract
 * @author The Swarm Authors
 * @dev Implements a Schelling Co-ordination game to form consensus around
    the Reserve Commitment hash. This takes
 * place in three phases: _commit_, _reveal_ and _claim_.
 *
 * A node, upon establishing that it _isParticipatingInUpcomingRound_, i.e.
    it's overlay falls within proximity order
 * of its reported depth with the _currentRoundAnchor_, prepares a "reserve
    commitment hash" using the chunks
 * it currently stores in its reserve and calculates the "storage depth" (see
     Bee for details). These values, if calculated
 * honestly, and with the right chunks stored, should be the same for every
    node in a neighbourhood. This is the Schelling point.
 * Each eligible node can then use these values, together with a random,
    single use, secret  _revealNonce_ and their
 * _overlay_ as the pre-image values for the obsfucated _commit_, using the
    _wrapCommit_ method.
 *
 * Once the _commit_ round has elapsed, participating nodes must provide the
    values used to calculate their obsfucated
 * _commit_ hash, which, once verified for correctness and proximity to the
    anchor are retained in the _currentReveals_.
 * Nodes that have commited but do not reveal the correct values used to
    create the pre-image will have their stake
 * "frozen" for a period of rounds proportional to their reported depth.
 *
 * During the _reveal_ round, randomness is updated after every successful
```

```solidity
reveal. Once the reveal round is concluded,
 * the _currentRoundAnchor_ is updated and users can determine if they will be eligible their overlay will be eligible
 * for the next commit phase using _isParticipatingInUpcomingRound_.
 *
 * When the _reveal_ phase has been concluded, the claim phase can begin. At this point, the truth teller and winner
 * are already determined. By calling _isWinner_, an applicant node can run the relevant logic to determine if they have
 * been selected as the beneficiary of this round. When calling _claim_, the current pot from the PostageStamp contract
 * is withdrawn and transferred to that beneficiaries address. Nodes that have revealed values that differ from the truth,
 * have their stakes "frozen" for a period of rounds proportional to their reported depth.
 */
contract Redistribution is AccessControl, Pausable {
    // An eligible user may commit to an _obfuscatedHash_ during the commit phase...
    struct Commit {
        bytes32 overlay;
        address owner;
        uint256 stake;
        bytes32 obfuscatedHash;
        bool revealed;
        uint256 revealIndex;
    }
    // ...then provide the actual values that are the constituents of the pre-image of the _obfuscatedHash_
    // during the reveal phase.
    struct Reveal {
        address owner;
        bytes32 overlay;
        uint256 stake;
        uint256 stakeDensity;
        bytes32 hash;
        uint8 depth;
    }

    // The address of the linked PostageStamp contract.
    PostageStamp public PostageContract;
    // The address of the linked PriceOracle contract.
    PriceOracle public OracleContract;
    // The address of the linked Staking contract.
    StakeRegistry public Stakes;

    // Commits for the current round.
    Commit[] public currentCommits;
    // Reveals for the current round.
    Reveal[] public currentReveals;

    // Role allowed to pause.
    bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

    uint256 public penaltyMultiplierDisagreement = 1;
    uint256 public penaltyMultiplierNonRevealed = 2;

    // Maximum value of the keccack256 hash.
    bytes32 MaxH = bytes32(0x00000000000000000000000000000000ffffffffffffffffffffffffffffffff);

    // The current anchor that being processed for the reveal and claim phases of the round.
    bytes32 currentRevealRoundAnchor;

    // The current random value from which we will random.
    // inputs for selection of the truth teller and beneficiary.
    bytes32 seed;

    // The miniumum stake allowed to be staked using the Staking contract.
    uint256 public minimumStake = 100000000000000000;

    // The number of the currently active round phases.
    uint256 public currentCommitRound;
    uint256 public currentRevealRound;
    uint256 public currentClaimRound;

    // The length of a round in blocks.
    uint256 public roundLength = 152;


    // The reveal of the winner of the last round.
    Reveal public winner;

    /**
    * @dev Pause the contract. The contract is provably stopped by renouncing
     the pauser role and the admin role after pausing, can only be called by the `PAUSER`
     */
    function pause() public {
        require(hasRole(PAUSER_ROLE, msg.sender), "only pauser can pause");
        _pause();
    }

    /**
     * @dev Unpause the contract, can only be called by the pauser when paused
     */
    function unPause() public {
        require(hasRole(PAUSER_ROLE, msg.sender), "only pauser can unpause");
        _unpause();
    }

    /**
     * @param staking the address of the linked Staking contract.
     * @param postageContract the address of the linked PostageStamp contract.
     * @param oracleContract the address of the linked PriceOracle contract.
     */
    constructor(address staking, address postageContract, address
        oracleContract) {
        Stakes = StakeRegistry(staking);
        PostageContract = PostageStamp(postageContract);
        OracleContract = PriceOracle(oracleContract);
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _setupRole(PAUSER_ROLE, msg.sender);
    }

    /**
     * @dev Emitted when the winner of a round is selected in the claim phase
     */
    event WinnerSelected(Reveal winner);

    /**
     * @dev Emitted when the truth oracle of a round is selected in the claim
         phase.
     */
    event TruthSelected(bytes32 hash, uint8 depth);

    // Next two events to be removed after testing phase pending some other
        usefulness being found.
    /**
     * @dev Emits the number of commits being processed by the claim phase.
     */
    event CountCommits(uint256 _count);

    /**
     * @dev Emits the number of reveals being processed by the claim phase.
     */
    event CountReveals(uint256 _count);

    /**
     * @dev Logs that an overlay has committed
     */
    event Committed(uint256 roundNumber, bytes32 overlay);
    /**
     * @dev Emit from Postagestamp contract valid chunk count at the end of
         claim
     */
    event ChunkCount(uint256 validChunkCount);

    /**
     * @dev Bytes32 anhor of current reveal round
     */
    event CurrentRevealAnchor(uint256 roundNumber, bytes32 anchor);

    /**
     * @dev Logs that an overlay has revealed
     */
    event Revealed(
        uint256 roundNumber,
        bytes32 overlay,
        uint256 stake,
        uint256 stakeDensity,
        bytes32 reserveCommitment,
        uint8 depth
    );

    /**
     * @notice Set freezing parameters
     */
    function setFreezingParams(uint256 _penaltyMultiplierDisagreement,
        uint256 _penaltyMultiplierNonRevealed) external {
        require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender), "caller is not the
            admin");
        penaltyMultiplierDisagreement = _penaltyMultiplierDisagreement;
        penaltyMultiplierNonRevealed = _penaltyMultiplierNonRevealed;
    }

    /**
     * @notice The number of the current round.
     */
    function currentRound() public view returns (uint256) {
        return (block.number / roundLength);
    }

    /**
     * @notice Returns true if current block is during commit phase.
     */
    function currentPhaseCommit() public view returns (bool) {
        if (block.number % roundLength < roundLength / 4) {
            return true;
        }
        return false;
    }

    /**
     * @notice Returns true if current block is during reveal phase.
     */
    function currentPhaseReveal() public view returns (bool) {
        uint256 number = block.number % roundLength;
        if (number >= roundLength / 4 && number < roundLength / 2) {
            return true;
        }
        return false;
    }

    /**
     * @notice Returns true if current block is during claim phase.
     */
    function currentPhaseClaim() public view returns (bool) {
        if (block.number % roundLength >= roundLength / 2) {
            return true;
        }
        return false;
    }

    /**
     * @notice Returns true if current block is during reveal phase.
     */
    function currentRoundReveals() public view returns (Reveal[] memory) {
        require(currentPhaseClaim(), "not in claim phase");
        uint256 cr = currentRound();
        require(cr == currentRevealRound, "round received no reveals");
        return currentReveals;
    }

    /**
     * @notice Begin application for a round if eligible. Commit a hashed value for which the pre-image will be
     * subsequently revealed.
     * @dev If a node's overlay is _inProximity_(_depth_) of the _currentRoundAnchor_, that node may compute an
     * _obfuscatedHash_ by providing their _overlay_, reported storage _depth_, reserve commitment _hash_ and a
     * randomly generated, and secret _revealNonce_ to the _wrapCommit_ method.
     * @param _obfuscatedHash The calculated hash resultant of the required pre-image values.
     * @param _overlay The overlay referenced in the pre-image. Must be staked by at least the minimum value,
     * and be derived from the same key pair as the message sender.
     */
    function commit(bytes32 _obfuscatedHash, bytes32 _overlay, uint256 _roundNumber) external whenNotPaused {
        require(currentPhaseCommit(), "not in commit phase");
        require(block.number % roundLength != (roundLength / 4) - 1, "can not commit in last block of phase");
        uint256 cr = currentRound();
        require(cr <= _roundNumber, "commit round over");
        require(cr >= _roundNumber, "commit round not started yet");

        uint256 nstake = Stakes.stakeOfOverlay(_overlay);
        require(nstake >= minimumStake, "stake must exceed minimum");
        require(Stakes.ownerOfOverlay(_overlay) == msg.sender, "owner must match sender");


        require(
            Stakes.lastUpdatedBlockNumberOfOverlay(_overlay) < block.number - 2 * roundLength,
            "must have staked 2 rounds prior"
        );

        // if we are in a new commit phase, reset the array of commits and
        // set the currentCommitRound to be the current one
        if (cr != currentCommitRound) {
            delete currentCommits;
            currentCommitRound = cr;
        }

        uint256 commitsArrayLength = currentCommits.length;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            require(currentCommits[i].overlay != _overlay, "only one commit
                each per round");
        }

        currentCommits.push(
            Commit({
                owner: msg.sender,
                overlay: _overlay,
                stake: nstake,
                obfuscatedHash: _obfuscatedHash,
                revealed: false,
                revealIndex: 0
            })
        );

        emit Committed(_roundNumber, _overlay);
    }

    /**
     * @notice Returns the current random seed which is used to determine
         later utilised random numbers.
     * If rounds have elapsed without reveals, hash the seed with an
         incremented nonce to produce a new
     * random seed and hence a new round anchor.
     */
    function currentSeed() public view returns (bytes32) {
        uint256 cr = currentRound();
        bytes32 currentSeedValue = seed;

        if (cr > currentRevealRound + 1) {
            uint256 difference = cr - currentRevealRound - 1;
            currentSeedValue = keccak256(abi.encodePacked(currentSeedValue,
                difference));
        }

        return currentSeedValue;
    }

    /**
     * @notice Returns the seed which will become current once the next
         commit phase begins.
     * Used to determine what the next round's anchor will be.
     */
    function nextSeed() public view returns (bytes32) {
        uint256 cr = currentRound() + 1;
        bytes32 currentSeedValue = seed;

        if (cr > currentRevealRound + 1) {
            uint256 difference = cr - currentRevealRound - 1;
            currentSeedValue = keccak256(abi.encodePacked(currentSeedValue,
                difference));
        }

        return currentSeedValue;
    }

    /**
     * @notice Updates the source of randomness. Uses block.difficulty in pre
         -merge chains, this is substituted
     * to block.prevrandao in post merge chains.
     */
    function updateRandomness() private {
        seed = keccak256(abi.encode(seed, block.prevrandao));
    }

    function nonceBasedRandomness(bytes32 nonce) private {
        seed = seed ^ nonce;
    }

    /**
     * @notice Returns true if an overlay address _A_ is within proximity
         order _minimum_ of _B_.
     * @param A An overlay address to compare.
     * @param B An overlay address to compare.
     * @param minimum Minimum proximity order.
     */
    function inProximity(bytes32 A, bytes32 B, uint8 minimum) public pure
        returns (bool) {
        if (minimum == 0) {
            return true;
        }
        return uint256(A ^ B) < uint256(2 ** (256 - minimum));
    }

    /**
     * @notice Hash the pre-image values to the obsfucated hash.
     * @dev _revealNonce_ must be randomly generated, used once and kept
         secret until the reveal phase.
     * @param _overlay The overlay address of the applicant.
     * @param _depth The reported depth.
     * @param _hash The reserve commitment hash.
     * @param revealNonce A random, single use, secret nonce.
     */
    function wrapCommit(
        bytes32 _overlay,
        uint8 _depth,
        bytes32 _hash,
        bytes32 revealNonce
    ) public pure returns (bytes32) {
        return keccak256(abi.encodePacked(_overlay, _depth, _hash,
            revealNonce));
    }


    // build this in revealL
    bytes32[] currentRevealHashes;
    mapping(bytes32 => uint256) currentRevealToStake;

    function _resetRevealToStake() internal {
    for (uint256 i = 0; i < currentRevealHashes.length; i++) {
        delete currentRevealToStake[currentRevealHashes[i]]; // Reset the
            value to 0
    }
    delete currentRevealHashes;
    }


    /**
     * @notice Reveal the pre-image values used to generate commit provided
         during this round's commit phase.
     * @param _overlay The overlay address of the applicant.
     * @param _depth The reported depth.
     * @param _hash The reserve commitment hash.
     * @param _revealNonce The nonce used to generate the commit that is
         being revealed.
     */
    function reveal(bytes32 _overlay, uint8 _depth, bytes32 _hash, bytes32
        _revealNonce) external whenNotPaused {
        require(currentPhaseReveal(), "not in reveal phase");

        uint256 cr = currentRound();

        require(cr == currentCommitRound, "round received no commits");
        if (cr != currentRevealRound) {
            //currentRevealRoundAnchor = currentRoundAnchor();
            //edit
            currentRevealRoundAnchor = currentRoundAnchorValue;

            delete currentReveals;
            _resetRevealToStake();
            currentRevealRound = cr;
            emit CurrentRevealAnchor(cr, currentRevealRoundAnchor);
            updateRandomness();
        }

        bytes32 commitHash = wrapCommit(_overlay, _depth, _hash, _revealNonce
            );

        uint256 commitsArrayLength = currentCommits.length;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            if (currentCommits[i].overlay == _overlay && commitHash ==
                currentCommits[i].obfuscatedHash) {
                //consider wether i Should bother with this check //get
                    around this with 0 depth
                require(
                    inProximity(currentCommits[i].overlay,
                        currentRevealRoundAnchor, _depth),
                    "anchor out of self reported depth"
                );
                //check can only revealed once
                require(currentCommits[i].revealed == false, "participant
                    already revealed");
                currentCommits[i].revealed = true;
                currentCommits[i].revealIndex = currentReveals.length;

                //build reveal map
               currentRevealHashes.push(_hash);
               if (currentRevealToStake[_hash] == 0) {
                    currentRevealToStake[_hash] = currentCommits[i].stake *
                        uint256(2 ** _depth);
                } else {
                    currentRevealToStake[_hash] += currentCommits[i].stake *
                        uint256(2 ** _depth);
                }

                //emit stakeSetToHash(currentRevealToStake[
                    currentRevealHashes[i]]);
                currentReveals.push(
                    Reveal({
                        owner: currentCommits[i].owner,
                        overlay: currentCommits[i].overlay,
                        stake: currentCommits[i].stake,
                        stakeDensity: currentCommits[i].stake * uint256(2 **
                            _depth),
                        hash: _hash,
                        depth: _depth
                    })
                );

                nonceBasedRandomness(_revealNonce);

                emit Revealed(
                    cr,
                    currentCommits[i].overlay,
                    currentCommits[i].stake,
                    currentCommits[i].stake * uint256(2 ** _depth),
                    _hash,
                    _depth
                );

                return;
            }
        }

        require(false, "no matching commit or hash");
    }






    /**
     * @notice Determine if a the owner of a given overlay will be the
         beneficiary of the claim phase.
     * @param _overlay The overlay address of the applicant.
     */
    function isWinner(bytes32 _overlay) public view returns (bool) {
        require(currentPhaseClaim(), "winner not determined yet");

        uint256 cr = currentRound();

        require(cr == currentRevealRound, "round received no reveals");
        require(cr > currentClaimRound, "round already received successful
            claim");

        string memory truthSelectionAnchor = currentTruthSelectionAnchor();

        uint256 currentSum;
        uint256 currentWinnerSelectionSum;
        bytes32 winnerIs;
        bytes32 randomNumber;

        bytes32 truthRevealedHash;
        uint8 truthRevealedDepth;

        uint256 commitsArrayLength = currentCommits.length;

        uint256 revIndex;
        uint256 k = 0;

        //new
        uint256 currentMaxReveal = 0;

        //can remove truth selection testing gas cost
        for (uint256 i = 0; i < commitsArrayLength; i++) {
            if (currentCommits[i].revealed) {
                revIndex = currentCommits[i].revealIndex;
                currentSum += currentReveals[revIndex].stakeDensity;
                randomNumber = keccak256(abi.encodePacked(
                    truthSelectionAnchor, k));

                //altered
                if (currentMaxReveal < currentRevealToStake[
                    currentRevealHashes[revIndex]] ) {
                    currentMaxReveal = currentRevealToStake[
                        currentRevealHashes[revIndex]];
                }

                if (
                    uint256(randomNumber & MaxH) * currentSum <
                    currentReveals[revIndex].stakeDensity * (uint256(MaxH) +
                        1)
                ) {
                    truthRevealedHash = currentReveals[revIndex].hash;
```

```solidity
                    truthRevealedDepth = currentReveals[revIndex].depth;
                }

                k++;
            }
        }
        uint256 bankStakeDensity = currentSum - currentMaxReveal;

        k = 0;

        string memory winnerSelectionAnchor = currentWinnerSelectionAnchor();
        //altered
        for (uint256 i = 0; i < commitsArrayLength+1; i++) {
            revIndex = currentCommits[i].revealIndex;
            if (currentCommits[i].revealed) {
            // if (
            //     currentCommits[i].revealed &&
            //     truthRevealedHash == currentReveals[revIndex].hash &&
            //     truthRevealedDepth == currentReveals[revIndex].depth
            // ) {
                currentWinnerSelectionSum += currentReveals[revIndex].
                    stakeDensity;
                randomNumber = keccak256(abi.encodePacked(
                    winnerSelectionAnchor, k));

                if (
                    uint256(randomNumber & MaxH) * currentWinnerSelectionSum
                        <
                    currentReveals[revIndex].stakeDensity * (uint256(MaxH) +
                        1)
                ) {
                    winnerIs = currentReveals[revIndex].overlay;
                }

                k++;
            }
            //new
            if (i >= commitsArrayLength){
                currentWinnerSelectionSum += bankStakeDensity;
                randomNumber = keccak256(abi.encodePacked(
                    winnerSelectionAnchor, k));

                 if(
                            uint256(randomNumber & MaxH) *
                                currentWinnerSelectionSum <
                            bankStakeDensity* (uint256(MaxH) + 1)
                        ) {
                           winnerIs = keccak256("BANK");
            }

        }
        }

        return (winnerIs == _overlay);
```

```
    }

    /**
     * @notice Determine if a the owner of a given overlay can participate in the upcoming round.
     * @param overlay The overlay address of the applicant.
     * @param depth The storage depth the applicant intends to report.
     */
    function isParticipatingInUpcomingRound(bytes32 overlay, uint8 depth) public view returns (bool) {
        require(currentPhaseClaim() || currentPhaseCommit(), "not determined for upcoming round yet");
        require(
            Stakes.lastUpdatedBlockNumberOfOverlay(overlay) < block.number - 2 * roundLength,
            "stake updated recently"
        );
        require(Stakes.stakeOfOverlay(overlay) >= minimumStake, "stake amount does not meet minimum");
        return inProximity(overlay, currentRoundAnchor(), depth);
    }

    /**
     * @notice The random value used to choose the selected truth teller.
     */
    function currentTruthSelectionAnchor() private view returns (string memory) {
        require(currentPhaseClaim(), "not determined for current round yet");
        uint256 cr = currentRound();
        require(cr == currentRevealRound, "round received no reveals");

        return string(abi.encodePacked(seed, "0"));
    }

    /**
     * @notice The random value used to choose the selected beneficiary.
     */
    function currentWinnerSelectionAnchor() private view returns (string memory) {
        require(currentPhaseClaim(), "not determined for current round yet");
        uint256 cr = currentRound();
        require(cr == currentRevealRound, "round received no reveals");

        return string(abi.encodePacked(seed, "1"));
    }

    /**
     * @notice The anchor used to determine eligibility for the current round.
     * @dev A node must be within proximity order of less than or equal to the storage depth they intend to report.
     */
    function currentRoundAnchor() public view returns (bytes32 returnVal) {
        uint256 cr = currentRound();

        if (currentPhaseCommit() || (cr > currentRevealRound && !currentPhaseClaim())) {
            return currentSeed();
        }

        if (currentPhaseReveal() && cr == currentRevealRound) {
            require(false, "can't return value after first reveal");
        }

        if (currentPhaseClaim()) {
            return nextSeed();
        }
    }

    //edited by me for testing
    bytes32 public currentRoundAnchorValue;

    function setCurrentRoundAnchor(bytes32 _value) external {
        currentRoundAnchorValue = _value;
    }

    string public currentTruthSelectionAnchorValue;

    function setCurrentTruthSelectionAnchor(string memory _value) external {
        currentTruthSelectionAnchorValue = _value;
    }



    /**
     * @notice Conclude the current round by identifying the selected truth teller and beneficiary.
     * @dev
     */
    function claim() external whenNotPaused {
        require(currentPhaseClaim(), "not in claim phase");

        uint256 cr = currentRound();

        require(cr == currentRevealRound, "round received no reveals");
        require(cr > currentClaimRound, "round already received successful claim");

        //string memory truthSelectionAnchor = currentTruthSelectionAnchor();
        //edit
        string memory truthSelectionAnchor = currentTruthSelectionAnchorValue;

        uint256 currentSum;
        uint256 currentWinnerSelectionSum;
        bytes32 randomNumber;
        uint256 randomNumberTrunc;

        bytes32 truthRevealedHash;
        uint8 truthRevealedDepth;

        uint256 commitsArrayLength = currentCommits.length;
        uint256 revealsArrayLength = currentReveals.length;

        emit CountCommits(commitsArrayLength);
        emit CountReveals(revealsArrayLength);

        uint256 revIndex;
        uint256 k = 0;

        // find the reveal with the max
        uint256 currentMaxReveal;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            if (currentCommits[i].revealed) {
                revIndex = currentCommits[i].revealIndex;
                currentSum += currentReveals[revIndex].stakeDensity;
                randomNumber = keccak256(abi.encodePacked(
                    truthSelectionAnchor, k));

                randomNumberTrunc = uint256(randomNumber & MaxH);

                // Find the maximum stake reveal
                if (currentMaxReveal < currentRevealToStake[
                    currentRevealHashes[revIndex]] ) {
                    currentMaxReveal = currentRevealToStake[
                        currentRevealHashes[revIndex]];
                }



                // question is whether randomNumber / MaxH < probability
                // where probability is stakeDensity / currentSum
                // to avoid resorting to floating points all divisions should
                     be
                // simplified with multiplying both sides (as long as divisor
                     > 0)
                // randomNumber / (MaxH + 1) < stakeDensity / currentSum
                // ( randomNumber / (MaxH + 1) ) * currentSum < stakeDensity
                // randomNumber * currentSum < stakeDensity * (MaxH + 1)
                if (randomNumberTrunc * currentSum < currentReveals[revIndex
                    ].stakeDensity * (uint256(MaxH) + 1)) {
                    truthRevealedHash = currentReveals[revIndex].hash;
                    truthRevealedDepth = currentReveals[revIndex].depth;
                }

                k++;
            }
        }


        uint256 bankStakeDensity = currentSum - currentMaxReveal;
```

```

       //currentSum += bankStakeDensity;
        //create dummy reveal for bank
        emit maxStakeEmitted(currentMaxReveal,bankStakeDensity);
        emit TruthSelected(truthRevealedHash, truthRevealedDepth);

        k = 0;

        string memory winnerSelectionAnchor = currentWinnerSelectionAnchor();

        for (uint256 i = 0; i < commitsArrayLength+1; i++) {
            if (i<commitsArrayLength){
                revIndex = currentCommits[i].revealIndex;
                if (currentCommits[i].revealed) {
                    // if (
                    //
                        //truthRevealedHash == currentReveals[revIndex].hash &&
                        //truthRevealedDepth == currentReveals[revIndex].depth
                    // )
                    //{
                        currentWinnerSelectionSum += currentReveals[revIndex].stakeDensity;
                        randomNumber = keccak256(abi.encodePacked(winnerSelectionAnchor, k));

                        randomNumberTrunc = uint256(randomNumber & MaxH);

                        if (
                            // randomNumberTrunc * currentWinnerSelectionSum <
                            // currentReveals[revIndex].stakeDensity * (uint256(MaxH) + 1)
                             randomNumberTrunc * currentWinnerSelectionSum <
                            currentReveals[revIndex].stakeDensity * (uint256(MaxH) + 1)
                        ) {
                            winner = currentReveals[revIndex];
                        }

                        k++;
                        //} else {
                //         Stakes.freezeDeposit(
                //             currentReveals[revIndex].overlay,
                //             penaltyMultiplierDisagreement * roundLength * uint256(2 ** truthRevealedDepth)
                //         );
                //         // slash ph5
                //     }
                } else {
                    // slash in later phase
                    // Stakes.slashDeposit(currentCommits[i].overlay, currentCommits[i].stake);
                    Stakes.freezeDeposit(
                        currentCommits[i].overlay,
                        penaltyMultiplierNonRevealed * roundLength * uint256(2 ** truthRevealedDepth)
                    );
                    continue;
                }
                  //  }

            }
            //new
            if (i >= commitsArrayLength){
                randomNumber = keccak256(abi.encodePacked(winnerSelectionAnchor, k+1));
                randomNumberTrunc = uint256(randomNumber & MaxH);

                currentWinnerSelectionSum += bankStakeDensity;
                 if(
                            randomNumberTrunc * currentWinnerSelectionSum <
                            bankStakeDensity* (uint256(MaxH) + 1)
                        ) {
                           winner = Reveal({
                         owner: address(this),
                         overlay: keccak256("BANK"),
                         stake: bankStakeDensity,
                         stakeDensity: bankStakeDensity,
                         hash: keccak256("bankhash"),
                         depth: 0
                     });

                        }

            }
        }

        if(winner.overlay == bytes32(0)){
             winner = Reveal({
                         owner: address(this),
                         overlay: keccak256("BANK"),
                         stake: bankStakeDensity,
                         stakeDensity: bankStakeDensity,
                         hash: keccak256("bankhash"),
                         depth: 0
                     });
        }

        emit WinnerSelected(winner);
        //do not pay ban, IE leave pot for next round
        if (winner.overlay != keccak256("BANK")){
                    PostageContract.withdraw(winner.owner);
        }

        emit ChunkCount(PostageContract.validChunkCount());

        OracleContract.adjustPrice(uint256(k));

        currentClaimRound = cr;
    }

    //functions and events to help testing
    function setCurrentClaimRound(uint256 newRound) external {
        require(hasRole(PAUSER_ROLE, msg.sender), "only pauser can pause");

        currentClaimRound = newRound;
    }
     function getCurrentClaimRound() public view returns (uint256) {
        return currentClaimRound;
    }

    event maxStakeEmitted(uint256 indexed mStake, uint256 indexed bStake);

    event emitNumber(uint256 indexed number);


    }
```

Listing B.1: Redistribution contract bank

Appendix C

Alpha solution contract code

```
    // SPDX-License-Identifier: BSD-3-Clause
pragma solidity ^0.8.1;
import "./@openzeppelin/contracts/access/AccessControl.sol";
import "./@openzeppelin/contracts/security/Pausable.sol";
import "./PostageStamp.sol";
import "./PriceOracle.sol";
import "./Staking.sol";

/**
 * @title Redistribution contract
 * @author The Swarm Authors
 * @dev Implements a Schelling Co-ordination game to form consensus around
    the Reserve Commitment hash. This takes
 * place in three phases: _commit_, _reveal_ and _claim_.
 *
 * A node, upon establishing that it _isParticipatingInUpcomingRound_, i.e.
    it's overlay falls within proximity order
 * of its reported depth with the _currentRoundAnchor_, prepares a "reserve
    commitment hash" using the chunks
 * it currently stores in its reserve and calculates the "storage depth" (see
     Bee for details). These values, if calculated
 * honestly, and with the right chunks stored, should be the same for every
    node in a neighbourhood. This is the Schelling point.
 * Each eligible node can then use these values, together with a random,
    single use, secret  _revealNonce_ and their
 * _overlay_ as the pre-image values for the obsfucated _commit_, using the
    _wrapCommit_ method.
 *
 * Once the _commit_ round has elapsed, participating nodes must provide the
    values used to calculate their obsfucated
 * _commit_ hash, which, once verified for correctness and proximity to the
    anchor are retained in the _currentReveals_.
 * Nodes that have commited but do not reveal the correct values used to
    create the pre-image will have their stake
 * "frozen" for a period of rounds proportional to their reported depth.
 *
 * During the _reveal_ round, randomness is updated after every successful
```

```
reveal. Once the reveal round is concluded,
 * the _currentRoundAnchor_ is updated and users can determine if they will be eligible their overlay will be eligible
 * for the next commit phase using _isParticipatingInUpcomingRound_.
 *
 * When the _reveal_ phase has been concluded, the claim phase can begin. At this point, the truth teller and winner
 * are already determined. By calling _isWinner_, an applicant node can run the relevant logic to determine if they have
 * been selected as the beneficiary of this round. When calling _claim_, the current pot from the PostageStamp contract
 * is withdrawn and transferred to that beneficiaries address. Nodes that have revealed values that differ from the truth,
 * have their stakes "frozen" for a period of rounds proportional to their reported depth.
 */
contract Redistribution2 is AccessControl, Pausable {
    // An eligible user may commit to an _obfuscatedHash_ during the commit phase...
    struct Commit {
        bytes32 overlay;
        address owner;
        uint256 stake;
        bytes32 obfuscatedHash;
        bool revealed;
        uint256 revealIndex;
    }
    // ...then provide the actual values that are the constituents of the pre-image of the _obfuscatedHash_
    // during the reveal phase.
    struct Reveal {
        address owner;
        bytes32 overlay;
        uint256 stake;
        uint256 stakeDensity;
        bytes32 hash;
        uint8 depth;
    }

    // The address of the linked PostageStamp contract.
    PostageStamp public PostageContract;
    // The address of the linked PriceOracle contract.
    PriceOracle public OracleContract;
    // The address of the linked Staking contract.
    StakeRegistry public Stakes;

    // Commits for the current round.
    Commit[] public currentCommits;
    // Reveals for the current round.
    Reveal[] public currentReveals;

    // Role allowed to pause.
    bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

    uint256 public penaltyMultiplierDisagreement = 1;
    uint256 public penaltyMultiplierNonRevealed = 2;

    // Maximum value of the keccack256 hash.
    //bytes32 MaxH = bytes32(0x00000000000000000000000000000000ffffffffffffffffffffffffffffffff);
    //edit to avoid arithmetic overflow
    bytes32 MaxH = bytes32(0x000000000000000000000000000000000000000000ffffffffffffffffffffff);


    // The current anchor that being processed for the reveal and claim phases of the round.
    bytes32 currentRevealRoundAnchor;

    // The current random value from which we will random.
    // inputs for selection of the truth teller and beneficiary.
    bytes32 seed;

    // The miniumum stake allowed to be staked using the Staking contract.
    uint256 public minimumStake = 100000000000000000;

    // The number of the currently active round phases.
    uint256 public currentCommitRound;
    uint256 public currentRevealRound;
    uint256 public currentClaimRound;

    // The length of a round in blocks.
    uint256 public roundLength = 152;

    //tuning parameter for redistribution game
    uint256 public alpha = 1;


    // The reveal of the winner of the last round.
    Reveal public winner;

    /**
    * @dev Pause the contract. The contract is provably stopped by renouncing
     the pauser role and the admin role after pausing, can only be called by the `PAUSER`
     */
    function pause() public {
        require(hasRole(PAUSER_ROLE, msg.sender), "only pauser can pause");
        _pause();
    }

    /**
     * @dev Unpause the contract, can only be called by the pauser when paused
     */
    function unPause() public {
        require(hasRole(PAUSER_ROLE, msg.sender), "only pauser can unpause");
        _unpause();
    }

    /**
     * @param staking the address of the linked Staking contract.
     * @param postageContract the address of the linked PostageStamp contract
         .
     * @param oracleContract the address of the linked PriceOracle contract.
     */
    constructor(address staking, address postageContract, address
        oracleContract) {
        Stakes = StakeRegistry(staking);
        PostageContract = PostageStamp(postageContract);
        OracleContract = PriceOracle(oracleContract);
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _setupRole(PAUSER_ROLE, msg.sender);
    }

    /**
     * @dev Emitted when the winner of a round is selected in the claim phase
     */
    event WinnerSelected(Reveal winner);

    /**
     * @dev Emitted when the truth oracle of a round is selected in the claim
          phase.
     */
    event TruthSelected(bytes32 hash, uint8 depth);

    // Next two events to be removed after testing phase pending some other
        usefulness being found.
    /**
     * @dev Emits the number of commits being processed by the claim phase.
     */
    event CountCommits(uint256 _count);

    /**
     * @dev Emits the number of reveals being processed by the claim phase.
     */
    event CountReveals(uint256 _count);

    /**
     * @dev Logs that an overlay has committed
     */
    event Committed(uint256 roundNumber, bytes32 overlay);
    /**
     * @dev Emit from Postagestamp contract valid chunk count at the end of
         claim
     */
    event ChunkCount(uint256 validChunkCount);

    /**
     * @dev Bytes32 anhor of current reveal round
     */
    event CurrentRevealAnchor(uint256 roundNumber, bytes32 anchor);
```

```solidity

    /**
     * @dev Logs that an overlay has revealed
     */
    event Revealed(
        uint256 roundNumber,
        bytes32 overlay,
        uint256 stake,
        uint256 stakeDensity,
        bytes32 reserveCommitment,
        uint8 depth
    );

    /**
     * @notice Set freezing parameters
     */
    function setFreezingParams(uint256 _penaltyMultiplierDisagreement,
        uint256 _penaltyMultiplierNonRevealed) external {
        require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender), "caller is not the
            admin");
        penaltyMultiplierDisagreement = _penaltyMultiplierDisagreement;
        penaltyMultiplierNonRevealed = _penaltyMultiplierNonRevealed;
    }

    /**
     * @notice The number of the current round.
     */
    function currentRound() public view returns (uint256) {
        return (block.number / roundLength);
    }

    /**
     * @notice Returns true if current block is during commit phase.
     */
    function currentPhaseCommit() public view returns (bool) {
        if (block.number % roundLength < roundLength / 4) {
            return true;
        }
        return false;
    }

    /**
     * @notice Returns true if current block is during reveal phase.
     */
    function currentPhaseReveal() public view returns (bool) {
        uint256 number = block.number % roundLength;
        if (number >= roundLength / 4 && number < roundLength / 2) {
            return true;
        }
        return false;
    }

    /**
     * @notice Returns true if current block is during claim phase.
     */
    function currentPhaseClaim() public view returns (bool) {
        if (block.number % roundLength >= roundLength / 2) {
            return true;
        }
        return false;
    }

    /**
     * @notice Returns true if current block is during reveal phase.
     */
    function currentRoundReveals() public view returns (Reveal[] memory) {
        require(currentPhaseClaim(), "not in claim phase");
        uint256 cr = currentRound();
        require(cr == currentRevealRound, "round received no reveals");
        return currentReveals;
    }

    /**
     * @notice Begin application for a round if eligible. Commit a hashed
         value for which the pre-image will be
     * subsequently revealed.
     * @dev If a node's overlay is _inProximity_(_depth_) of the
         _currentRoundAnchor_, that node may compute an
     * _obfuscatedHash_ by providing their _overlay_, reported storage
         _depth_, reserve commitment _hash_ and a
     * randomly generated, and secret _revealNonce_ to the _wrapCommit_
         method.
     * @param _obfuscatedHash The calculated hash resultant of the required
         pre-image values.
     * @param _overlay The overlay referenced in the pre-image. Must be
         staked by at least the minimum value,
     * and be derived from the same key pair as the message sender.
     */
    function commit(bytes32 _obfuscatedHash, bytes32 _overlay, uint256
        _roundNumber) external whenNotPaused {
        require(currentPhaseCommit(), "not in commit phase");
        require(block.number % roundLength != (roundLength / 4) - 1, "can not
             commit in last block of phase");
        uint256 cr = currentRound();
        require(cr <= _roundNumber, "commit round over");
        require(cr >= _roundNumber, "commit round not started yet");

        uint256 nstake = Stakes.stakeOfOverlay(_overlay);
        require(nstake >= minimumStake, "stake must exceed minimum");
        require(Stakes.ownerOfOverlay(_overlay) == msg.sender, "owner must
            match sender");

        require(
             Stakes.lastUpdatedBlockNumberOfOverlay(_overlay) < block.number
                 - 2 * roundLength,
             "must have staked 2 rounds prior"
        );

```

```solidity
        // if we are in a new commit phase, reset the array of commits and
        // set the currentCommitRound to be the current one
        if (cr != currentCommitRound) {
            delete currentCommits;
            currentCommitRound = cr;
        }

        uint256 commitsArrayLength = currentCommits.length;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            require(currentCommits[i].overlay != _overlay, "only one commit
                each per round");
        }

        currentCommits.push(
            Commit({
                owner: msg.sender,
                overlay: _overlay,
                stake: nstake,
                obfuscatedHash: _obfuscatedHash,
                revealed: false,
                revealIndex: 0
            })
        );

        emit Committed(_roundNumber, _overlay);
    }

    /**
     * @notice Returns the current random seed which is used to determine
         later utilised random numbers.
     * If rounds have elapsed without reveals, hash the seed with an
         incremented nonce to produce a new
     * random seed and hence a new round anchor.
     */
    function currentSeed() public view returns (bytes32) {
        uint256 cr = currentRound();
        bytes32 currentSeedValue = seed;

        if (cr > currentRevealRound + 1) {
            uint256 difference = cr - currentRevealRound - 1;
            currentSeedValue = keccak256(abi.encodePacked(currentSeedValue,
                difference));
        }

        return currentSeedValue;
    }

    /**
     * @notice Returns the seed which will become current once the next
         commit phase begins.
     * Used to determine what the next round's anchor will be.
     */
    function nextSeed() public view returns (bytes32) {
        uint256 cr = currentRound() + 1;
        bytes32 currentSeedValue = seed;

        if (cr > currentRevealRound + 1) {
            uint256 difference = cr - currentRevealRound - 1;
            currentSeedValue = keccak256(abi.encodePacked(currentSeedValue,
                difference));
        }

        return currentSeedValue;
    }

    /**
     * @notice Updates the source of randomness. Uses block.difficulty in pre
         -merge chains, this is substituted
     * to block.prevrandao in post merge chains.
     */
    function updateRandomness() private {
        seed = keccak256(abi.encode(seed, block.prevrandao));
    }

    function nonceBasedRandomness(bytes32 nonce) private {
        seed = seed ^ nonce;
    }

    /**
     * @notice Returns true if an overlay address _A_ is within proximity
         order _minimum_ of _B_.
     * @param A An overlay address to compare.
     * @param B An overlay address to compare.
     * @param minimum Minimum proximity order.
     */
    function inProximity(bytes32 A, bytes32 B, uint8 minimum) public pure
        returns (bool) {
        if (minimum == 0) {
            return true;
        }
        return uint256(A ^ B) < uint256(2 ** (256 - minimum));
    }

    /**
     * @notice Hash the pre-image values to the obsfucated hash.
     * @dev _revealNonce_ must be randomly generated, used once and kept
         secret until the reveal phase.
     * @param _overlay The overlay address of the applicant.
     * @param _depth The reported depth.
     * @param _hash The reserve commitment hash.
     * @param revealNonce A random, single use, secret nonce.
     */
    function wrapCommit(
        bytes32 _overlay,
        uint8 _depth,
        bytes32 _hash,
        bytes32 revealNonce
```

```solidity
    ) public pure returns (bytes32) {
        return keccak256(abi.encodePacked(_overlay, _depth, _hash,
            revealNonce));
    }


    // build this in reveaL
    bytes32[] currentRevealHashes;
    mapping(bytes32 => uint256) currentRevealToStake;

    function _resetRevealToStake() internal {
    for (uint256 i = 0; i < currentRevealHashes.length; i++) {
        delete currentRevealToStake[currentRevealHashes[i]]; // Reset the
            value to 0
    }
    delete currentRevealHashes;
    }




    /**
     * @notice Reveal the pre-image values used to generate commit provided
         during this round's commit phase.
     * @param _overlay The overlay address of the applicant.
     * @param _depth The reported depth.
     * @param _hash The reserve commitment hash.
     * @param _revealNonce The nonce used to generate the commit that is
         being revealed.
     */
    function reveal(bytes32 _overlay, uint8 _depth, bytes32 _hash, bytes32
        _revealNonce) external whenNotPaused {
        require(currentPhaseReveal(), "not in reveal phase");

        uint256 cr = currentRound();

        require(cr == currentCommitRound, "round received no commits");
        if (cr != currentRevealRound) {
            //currentRevealRoundAnchor = currentRoundAnchor();
            //edit
            currentRevealRoundAnchor = currentRoundAnchorValue;
            delete currentReveals;
             _resetRevealToStake();
            currentRevealRound = cr;
            emit CurrentRevealAnchor(cr, currentRevealRoundAnchor);
            updateRandomness();
        }

        bytes32 commitHash = wrapCommit(_overlay, _depth, _hash, _revealNonce
            );

        uint256 commitsArrayLength = currentCommits.length;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            if (currentCommits[i].overlay == _overlay && commitHash == currentCommits[i].obfuscatedHash) {
                require(
                    inProximity(currentCommits[i].overlay, currentRevealRoundAnchor, _depth),
                    "anchor out of self reported depth"
                );
                //check can only revealed once
                require(currentCommits[i].revealed == false, "participant already revealed");
                currentCommits[i].revealed = true;
                currentCommits[i].revealIndex = currentReveals.length;


                 //build reveal map
               currentRevealHashes.push(_hash);
               if (currentRevealToStake[_hash] == 0) {
                    currentRevealToStake[_hash] = currentCommits[i].stake * uint256(2 ** _depth);
                } else {
                    currentRevealToStake[_hash] += currentCommits[i].stake * uint256(2 ** _depth);
                }
                emit RevealToStakeMapUpdated( currentRevealToStake[_hash]);

                currentReveals.push(
                    Reveal({
                        owner: currentCommits[i].owner,
                        overlay: currentCommits[i].overlay,
                        stake: currentCommits[i].stake,
                        stakeDensity: currentCommits[i].stake * uint256(2 ** _depth),
                        hash: _hash,
                        depth: _depth
                    })
                );

                nonceBasedRandomness(_revealNonce);

                emit Revealed(
                    cr,
                    currentCommits[i].overlay,
                    currentCommits[i].stake,
                    currentCommits[i].stake * uint256(2 ** _depth),
                    _hash,
                    _depth
                );

                return;
            }
        }

        require(false, "no matching commit or hash");
    }
    //babylonian square root
    function sqrt(uint x) public pure returns (uint y) {
            if (x == 0) return 0;
            else if (x <= 3) return 1;

            uint z = (x + 1) / 2;
            y = x;
            while (z < y ) {
                y = z;
                z = (x / z + z) / 2;
            }
            return y;
        }



    function cbrt(uint x) public pure returns (uint y) {
            if (x == 0) return 0;
            uint8 k = 0;

            uint z = (x + 1) / 3;
            y = x;
            while (z < y) {
                y = z;
                z = (x / (z * z) + 2 * z) / 3;
            }
            return y;

        }


    /**
     * @notice Determine if a the owner of a given overlay will be the
         beneficiary of the claim phase.
     * @param _overlay The overlay address of the applicant.
     */
    function isWinner(bytes32 _overlay) public view returns (bool) {
        require(currentPhaseClaim(), "winner not determined yet");

        uint256 cr = currentRound();

        require(cr == currentRevealRound, "round received no reveals");
        require(cr > currentClaimRound, "round already received successful
            claim");

        string memory truthSelectionAnchor = currentTruthSelectionAnchor();

        uint256 currentSum;
        uint256 currentWinnerSelectionSum;
        bytes32 winnerIs;
        bytes32 randomNumber;

        bytes32 truthRevealedHash;
        uint8 truthRevealedDepth;

        uint256 commitsArrayLength = currentCommits.length;

        uint256 revIndex;
        uint256 k = 0;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            if (currentCommits[i].revealed) {
                revIndex = currentCommits[i].revealIndex;
                currentSum += currentReveals[revIndex].stakeDensity;
                randomNumber = keccak256(abi.encodePacked(
                    truthSelectionAnchor, k));

                if (
                    uint256(randomNumber & MaxH) * currentSum <
                    currentReveals[revIndex].stakeDensity * (uint256(MaxH) +
                        1)
                ) {
                    truthRevealedHash = currentReveals[revIndex].hash;
                    truthRevealedDepth = currentReveals[revIndex].depth;
                }

                k++;
            }
        }

        k = 0;

        string memory winnerSelectionAnchor = currentWinnerSelectionAnchor();

        randomNumber = keccak256(abi.encodePacked(winnerSelectionAnchor, k));

        uint256 currentWheelSliceStart = 0;
        uint256 currentWheelSliceEnd = 0;


        for (uint256 i = 0; i < commitsArrayLength; i++) {
            revIndex = currentCommits[i].revealIndex;
            if (
                currentCommits[i].revealed &&
                truthRevealedHash == currentReveals[revIndex].hash &&
                truthRevealedDepth == currentReveals[revIndex].depth
            ) {
                currentWheelSliceEnd =  currentWheelSliceStart + (
                    currentReveals[revIndex].stakeDensity*sqrt(
                    currentRevealToStake[currentReveals[revIndex].hash])*
                    alpha * (uint256(MaxH) + 1));

                uint256 randCalc = uint256(randomNumber & MaxH)*currentSum *
                    sqrt(currentSum) * alpha;


                if (
                        (currentWheelSliceStart <= randCalc) &&
                        randCalc <
                            currentWheelSliceEnd
                    ) {
                        winnerIs = currentReveals[revIndex].overlay;
                        break;
                     }
                    currentWheelSliceStart = currentWheelSliceEnd;
                    k++;
            }
        }

        if (winnerIs == bytes32(0)){
            winnerIs = keccak256("BANK");
        }

        return (winnerIs == _overlay);
    }

    /**
     * @notice Determine if a the owner of a given overlay can participate in the upcoming round.
     * @param overlay The overlay address of the applicant.
     * @param depth The storage depth the applicant intends to report.
     */
    function isParticipatingInUpcomingRound(bytes32 overlay, uint8 depth) public view returns (bool) {
        require(currentPhaseClaim() || currentPhaseCommit(), "not determined for upcoming round yet");
        require(
            Stakes.lastUpdatedBlockNumberOfOverlay(overlay) < block.number - 2 * roundLength,
            "stake updated recently"
        );
        require(Stakes.stakeOfOverlay(overlay) >= minimumStake, "stake amount does not meet minimum");
        return inProximity(overlay, currentRoundAnchor(), depth);
    }

    /**
     * @notice The random value used to choose the selected truth teller.
     */
    function currentTruthSelectionAnchor() private view returns (string memory) {
        require(currentPhaseClaim(), "not determined for current round yet");
        uint256 cr = currentRound();
        require(cr == currentRevealRound, "round received no reveals");

        return string(abi.encodePacked(seed, "0"));
    }

    /**
     * @notice The random value used to choose the selected beneficiary.
     */
    function currentWinnerSelectionAnchor() private view returns (string
        memory) {
        require(currentPhaseClaim(), "not determined for current round yet");
        uint256 cr = currentRound();
        require(cr == currentRevealRound, "round received no reveals");

        return string(abi.encodePacked(seed, "1"));
    }

    /**
     * @notice The anchor used to determine eligibility for the current round
        .
     * @dev A node must be within proximity order of less than or equal to
        the storage depth they intend to report.
     */
    function currentRoundAnchor() public view returns (bytes32 returnVal) {
        uint256 cr = currentRound();

        if (currentPhaseCommit() || (cr > currentRevealRound && !
            currentPhaseClaim())) {
            return currentSeed();
        }

        if (currentPhaseReveal() && cr == currentRevealRound) {
            require(false, "can't return value after first reveal");
        }

        if (currentPhaseClaim()) {
            return nextSeed();
        }
    }

//edited by me for testing
    bytes32 public currentRoundAnchorValue;

    function setCurrentRoundAnchor(bytes32 _value) external {
        currentRoundAnchorValue = _value;
    }

    string public currentTruthSelectionAnchorValue;

    function setCurrentTruthSelectionAnchor(string memory _value) external {
        currentTruthSelectionAnchorValue = _value;
    }

    /**
     * @notice Conclude the current round by identifying the selected truth
        teller and beneficiary.
     * @dev
     */
    function claim() external whenNotPaused {
        require(currentPhaseClaim(), "not in claim phase");

        uint256 cr = currentRound();

        require(cr == currentRevealRound, "round received no reveals");
        require(cr > currentClaimRound, "round already received successful claim");
        delete winner;
       //string memory truthSelectionAnchor = currentTruthSelectionAnchor();
        //edit
        string memory truthSelectionAnchor = currentTruthSelectionAnchorValue;

        uint256 currentSum;
        //uint256 currentWinnerSelectionSum;
        bytes32 randomNumber;
        uint256 randomNumberTrunc;

        bytes32 truthRevealedHash;
        uint8 truthRevealedDepth;

        uint256 commitsArrayLength = currentCommits.length;
        uint256 revealsArrayLength = currentReveals.length;

        emit CountCommits(commitsArrayLength);
        emit CountReveals(revealsArrayLength);

        uint256 revIndex;
        uint256 k = 0;

        //can remove if we forget about freezing
        for (uint256 i = 0; i < commitsArrayLength; i++) {
            if (currentCommits[i].revealed) {
                revIndex = currentCommits[i].revealIndex;
                currentSum += currentReveals[revIndex].stakeDensity;
                randomNumber = keccak256(abi.encodePacked(truthSelectionAnchor, k));

                randomNumberTrunc = uint256(randomNumber & MaxH);

                // question is whether randomNumber / MaxH < probability
                // where probability is stakeDensity / currentSum
                // to avoid resorting to floating points all divisions should be
                // simplified with multiplying both sides (as long as divisor > 0)
                // randomNumber / (MaxH + 1) < stakeDensity / currentSum
                // ( randomNumber / (MaxH + 1) ) * currentSum < stakeDensity
                // randomNumber * currentSum < stakeDensity * (MaxH + 1)
                if (randomNumberTrunc * currentSum < currentReveals[revIndex].stakeDensity * (uint256(MaxH) + 1)) {
                    truthRevealedHash = currentReveals[revIndex].hash;
                    truthRevealedDepth = currentReveals[revIndex].depth;
                }

                k++;
            }
        }

        emit TruthSelected(truthRevealedHash, truthRevealedDepth);

        k = 0;
        //need stake belonging to reveal
        string memory winnerSelectionAnchor = currentWinnerSelectionAnchor();


        randomNumber = keccak256(abi.encodePacked(winnerSelectionAnchor, k));

        uint256 currentWheelSliceStart = 0;
        uint256 currentWheelSliceEnd = 0;

        for (uint256 i = 0; i < commitsArrayLength; i++) {
            revIndex = currentCommits[i].revealIndex;
            if (currentCommits[i].revealed) {
                // if (
                //     truthRevealedHash == currentReveals[revIndex].hash &&
                //     truthRevealedDepth == currentReveals[revIndex].depth
                // ) {
                    //currentWinnerSelectionSum += currentReveals[revIndex].
                        stakeDensity;
                    //randomNumber = keccak256(abi.encodePacked(
                        winnerSelectionAnchor, k));

                    randomNumberTrunc = uint256(randomNumber & MaxH);

                    //need stop these from creating integer overflow

                    currentWheelSliceEnd =  currentWheelSliceStart + (
                        currentReveals[revIndex].stakeDensity*sqrt(
                        currentRevealToStake[currentReveals[revIndex].hash])
                        *alpha * (uint256(MaxH) + 1));

                    uint256 randCalc = randomNumberTrunc*currentSum *sqrt(
                        currentSum) * alpha;


                        //do initially with alpha ==1
                    if (
                        (currentWheelSliceStart <= randCalc) &&
                        randCalc <
                            currentWheelSliceEnd
                    ) {
                        winner = currentReveals[revIndex];
                        //breaking here might mess with some properties
                        break;
                     }
                    currentWheelSliceStart = currentWheelSliceEnd;
                    k++;
                // } else {
                //     Stakes.freezeDeposit(
                //          currentReveals[revIndex].overlay,
                //          penaltyMultiplierDisagreement * roundLength *
                    uint256(2 ** truthRevealedDepth)
                //      );
                    // slash ph5
               //  }
            } else {
                // slash in later phase
                // Stakes.slashDeposit(currentCommits[i].overlay,
                    currentCommits[i].stake);
                Stakes.freezeDeposit(
                    currentCommits[i].overlay,
                    penaltyMultiplierNonRevealed * roundLength * uint256(2 **
                         truthRevealedDepth)
                );
                continue;
            }
        }
        if (winner.overlay == bytes32(0)){
            winner = Reveal({
                          owner: address(this),
                          overlay: keccak256("BANK"),
                          stake: 0,
                          stakeDensity: 0,
                          hash: keccak256("bankhash"),
                          depth: 0
                     });
        }


        emit WinnerSelected(winner);

        //do not pay ban, IE leave pot for next round

        if (winner.overlay != keccak256("BANK") && winner.overlay != bytes32
            (0) ){
                    PostageContract.withdraw(winner.owner);
        }


        emit ChunkCount(PostageContract.validChunkCount());

        OracleContract.adjustPrice(uint256(k));

        currentClaimRound = cr;
    }


      //functions and events to help testing
    function setCurrentClaimRound(uint256 newRound) external {
        require(hasRole(PAUSER_ROLE, msg.sender), "only pauser can pause");

        currentClaimRound = newRound;
    }
```

```
     function getCurrentClaimRound() public view returns (uint256) {
        return currentClaimRound;
    }

    event RevealToStakeMapUpdated(uint256  indexed stake);

    event emitNumber(uint256 indexed number);




}
```

Listing C.1: Redistribution contract alpha

Bibliography

[1] Google, "Google drive," 2024. (accessed Jan. 9, 2024).

[2] Microsoft, "Microsoft onedrive," 2024. (accessed Jan. 9, 2024).

[3] A. Heinzman, "Google drive is mysteriously losing user data (updated)," 2023. (accessed Jan. 9, 2024).

[4] Swarm-team, "Future-proof storage," 2023.

[5] Filecoin, "How storage works," 2024. (accessed Jan. 30, 2024).

[6] Storj-labs, "Storj whitepaper v3," 2018. (accessed Jan. 30, 2024).

[7] Sia, "About renting on sia," 2024. (accessed Jan. 30, 2024).

[8] Ethereum-Swarm, "Storage incentives: The missing piece to make blockchains complete," 2023. (accessed Jan. 9, 2024).

[9] Ethereum-Swarm, "Host your website on swarm," 2024. (accessed Jan. 16, 2024).

[10] "Analyzing accessibility of wikipedia projects around the world," 2017. (accessed Jan. 16, 2024).

[11] Kiwix, "A new partnership to build a decentralized wikipedia," 2024. (accessed Jan. 16, 2024).

[12] S. J. Nielson, C. E. Spare, and D. S. Wallach, "Building better incentives for robustness in bittorrent," 2011.

[13] R. Nygaard, H. Meling, and J. I. Olsen, "Cost-effective data upkeep in decentralized storage systems," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, (New York, NY, USA), p. 165–173, Association for Computing Machinery, 2023.

[14] J. Guljaš, "Swarm scan," 2024. (accessed Feb. 9, 2024).

[15] V. H. Lakhani, L. Jehl, R. Hendriksen, and V. Estrada-Galiñanes, "Fair incentivization of bandwidth sharing in decentralized storage networks," in *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 39–44, 2022.

[16] K. H. Tjessem, "Simulating swarm's storage incentive," 2023.

[17] E. N. Tas and D. Boneh, "Cryptoeconomic security for data availability committees," 2023.

[18] C. Li, M. Xu, J. Zhang, H. Guo, and X. Cheng, "Sok: Decentralized storage network." Cryptology ePrint Archive, Paper 2024/258, 2024. `https://eprint.iacr.org/2024/258`.

[19] N. Zahed Benisi, M. Aminian, and B. Javadi, "Blockchain-based decentralized storage networks: A survey," *Journal of Network and Computer Applications*, vol. 162, p. 102656, 2020.

[20] V. Trón, *The Book of Swarm.* 2023. accessed Jan 9th 2024.

[21] IPFS-community, "libp2p," 2024. (accessed Jan. 16, 2024).

[22] Swarm-Team, "Swarm specification," 2023. (accessed Jan. 18, 2024).

[23] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems* (P. Druschel, F. Kaashoek, and A. Rowstron, eds.), (Berlin, Heidelberg), pp. 53–65, Springer Berlin Heidelberg, 2002.

[24] The-Swarm-Authors, "Redistribution contract," 2024. (accessed Jan. 16, 2024).

[25] P. Talwalkar, "Focal points (or schelling points): How we naturally organize in games of coordination," 2008. (accessed Jan. 30, 2024).

[26] Bee-Team, "Bee-scripts readsi," 2024. (accessed Feb. 9, 2023).

[27] Swarm-Team, "Neighbourhoods-swarm," 2024. (accessed Jan. 18, 2024).

[28] Wolfram Alpha, "epminput," 2024. Accessed on April 17, 2024.

[29] GeoGebra, "Geogebra classic," 2024. Accessed on April 17, 2024.

[30] Swarm-Team, "Swarm scan," 2024. (accessed Mar. 18, 2024).