University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2019 Open/~~Confidential~~
Author: Daniel Barati	Daniel Barati (signature of author)
Faculty supervisor: Leander Jehl	
Title of Master's thesis: Self-Emerging Proof-of-Storage Challenges Using Smart Contracts	
Credits: 30 ECTS	
Keywords: Proof-of-Storage • Distributed Systems Blockchain • Smart Contract • Ethereum	Number of pages: 85 + supplemental material/other: - Code included as link in Appenix - Experimental results included as tables in Appendix Stavanger, June 15 2019

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Self-Emerging Proof-of-Storage Challenges Using Smart Contracts

Master's Thesis in Computer Science
by

Daniel Barati

Supervisor

Leander Jehl

June 15, 2019

Abstract

Proof-of-Storage (PoS) is a collective term for protocols that allow proving data integrity and availability. There exist several PoS schemes. While they differ in detailed specifications, their common primary advantage is eliminating the need for trust between storage providers and data owners. However, there does not exist a mechanism to provide self-emerging delivery of requests for proof of storage, commonly known as challenges.

This paper presents a decentralized system for PoS using *self-emerging challenges* built on smart contract in the Ethereum platform. Self-emerging challenges provide an automated mechanism for ensuring integrity and persistence of data at chosen time intervals. The design employs participating nodes in the Ethereum blockchain, commonly referred to as peers, to store and route challenges to storage providers. The peers are compensated for their service by their respective employers. Data owners are enabled to schedule the time of emergence of a challenge to storage providers. Upon a received challenge, storage providers prove the integrity and persistence of data by responding correctly to the challenge. The design builds on the existing work of decentralized self-emerging data systems over Ethereum blockchain networks. We show that this work can be utilized for PoS and solve the problems that the incorporation and adaptation of this work raises.

We evaluate the proposed system based on several factors. We investigate the security of the system based on the different attacks that the participants may execute for exploitation. Moreover, we evaluate the attractiveness of participating in the system based on the gained remuneration by peers and the positive reputation gained by storage providers for proving the integrity of their clients' data. We also evaluate the expenses of data owners utilizing the proposed system based on the inherited costs of invoking smart contract functions in the Ethereum platform. Lastly, through analysis, we find that to minimize the total costs in the system, the number of employed peers should be restricted to one in each path. In other words, one peer to deliver a PoS challenge to the storage service provider. We show that this additionally improves the fairness of remuneration payout to peers and analyze how security is affected by always utilizing one peer in each path. We discover that this improves prevention against drop attacks, while it to some degree decreases the prevention of release-ahead attacks which we deem less critical. Through these analyses, we recognize that the benefits greatly outweigh the drawback, and we make a suggestion that data owners should select exactly one peer per path in their services.

Acknowledgements

I would like to extend my sincere gratitude to my supervisor Professor Leander Jehl for his guidance throughout this research work. The weekly meetings, discussions and encouragements have been much appreciated. I also would like to thank Professor Hein Meling and Rodrigo Saramago for their valuable insight and ideas.

I am grateful for the support of Elisabeth Bratli, my family and friends; both through this busy semester and otherwise in life.

Contents

Abbreviations

PoS	**P**roof-**o**f-**S**torage
P2P	**P**eer-to-**P**eer
EVM	**E**thereum **V**irtual **M**achine

Chapter 1

Introduction

The advances in networking technology and the rapid increase of digitalization in the everyday life of both industry and individuals have led to the use of remote storage solutions. The permanent availability of data is crucial, and ample amounts of data are produced every day. Therefore the need for storage at off-site locations arises [1]. Today, the amount of cloud storage providers are many, and they offer tailored services for different customers [2].

Data loss and data corruption are significant challenges and are typical results of management or hardware failures [3]. Both data loss and corruption can remain unknown until the data is accessed. This poses a problem since large amounts of data may rarely be accessed. Proving data integrity by retrieval is a poor method since it will greatly impact both the network and the local hard drive. Hence, alternative methods of proving data possession are required. These methods should limit the usage of network bandwidth and be computationally cheap.

Proof-of-Storage (PoS) describes protocols that allow a party to verify the integrity of remotely stored data [4]. In principle, a data owner issues a request for proof of storage, commonly known as a challenge, and the data storer proves data integrity by responding correctly to the challenge. A simple PoS scheme can be designed by probing. That is, the data owner stores different segments of their remotely stored data, and requests the data storer to retrieve the segments. There exist several other PoS schemes with various advantages and disadvantages using a variety of data structures and algorithms.

This thesis investigates PoS using self-emerging challenges, which provides an automated system for ensuring integrity and persistence of data. A promising method of implementing autonomous release of protected data at a certain time, without the need of manual interaction, is with the use of smart contracts as a trusted third party where the ground rules of the system are set. Smart contracts are distributed applications that run on blockchain platforms. This effectively means that they do not rely on a single point of failure. Therefore, employing such a smart contract implementation for a PoS system will enable the autonomous release of challenges without a single point of failure. We build on existing work of decentralized self-emerging data systems over Ethereum blockchain networks [5], and we show that this work can be utilized for PoS. Furthermore, we solve the challenges that the incorporation and adaptation of this work raises. We will also evaluate the security and Ethereum gas costs of the system.

To prove data integrity continuously, the PoS algorithm needs to be executed frequently. This means that the data owner should issue a challenge, requesting proof of storage at regular time intervals. By utilizing the PoS scheme using probing, the data owner needs to calculate and store the hash of the data at different indexes, creating pairs of challenges and corresponding correct answers before only relying on the services of the data storer. It can then issue the stored challenges to the data storer. Upon an issued challenge, the data storer should respond with an answer in order to prove the existence and integrity of the data.

A PoS system consisting of only a data owner and its data storer is impractical since it depends on the end user, the data owner, to deliver the challenges itself. Therefore, the system must be designed such that challenges are delivered to the data storer regardless of unavailability or failure at the data owner. The system needs to be able to offer both automatic delivery of challenges with definite time intervals, in addition to reception and verification of answers by the data storer. This enables the data owner to outsource the job of delivering challenges to the data storer and verifying the answers given by the data storer. Furthermore, in the case of trust-less decentralized storage systems where continuous data integrity needs to be proven, the incorporation of a mechanism where PoS challenges are frequently delivered to data storers is necessary. Provided that self-emerging challenges for PoS are offered in the system, data owners can be given the freedom of going offline and still frequently issuing challenges to their respective data storers.

It is possible to eliminate single point of failure by using a distributed scheme for this system. This can be done by employing peers in a distributed system to store and deliver the challenges on behalf of the data owner. Attracting peers to partake in the system can be done by offering remuneration for their services paid by their employer, that is the data owner. Additionally, the challenges must be verifiable by a trusted third party since the data owner should be able to issue the challenges and go offline. Therefore a third party must be able to receive challenges and corresponding answers from the data owner and deliver the challenges to the data storer at the appropriate time. Furthermore, the data storer needs to be able to prove the integrity of the data by providing answers upon the reception of challenges. For this, a third party could also be used to verify if the answers submitted by the data storer are correct.

The set of requirements of the system is given in the following list.

- Enable automatic delivery of multiple challenges with definite time intervals.
- Delivery of challenges regardless of failure or unavailability at the data owner.
- Correctly verify submitted answers to the challenges without the presence of the data owner in the process.
- Offer sufficient remuneration to partaking participants in the system for their service while still limiting the expenses of the data owner to be reasonable.

We build on a protocol proposed in [5] to benefit from the self-emerging data architecture. Additionally, we utilize its idea of employing nodes in the Ethereum platform, commonly known as peers, and using a smart contract as a trusted third party. However, incorporating their protocol into our system also raises some challenges.

In [5] the challenges of decentralizing self-emerging data release is solved by their proposed timed-release service protocol. The protocol utilizes a smart contract, which acts as a trusted third party, in order to set ground rules for the involved participants in the system. The addressed challenge in this protocol is mainly decentralizing the data release. It employs a set of Ethereum peers to form a path through the network in order to send a cryptographic key.

To build upon this protocol, we address new problems that arise as a consequence. Our proposed system, that is an extension of the timed-release service protocol for PoS with autonomous and continuous release of challenges over long periods of time, requires multiple paths of Ethereum peers for different PoS challenges. However, the existing protocol form only one path of peers. We redesign the smart contract to enable multiple paths while still maintaining the security of the system. Moreover, since costs in the system are determined by gas costs of function calls in the smart contract and remuneration to peers, we also discuss and evaluate the costs of the system as it is imperative to address the challenges related to setting the right remuneration and minimizing gas costs in order to make the system attractive for both data owners and peers. One of the most significant differences from the timed-release protocol is the difference in the roles of the participants. Since data owners and data storers are not directly comparable to the roles presented in the timed-release protocol, a number of changes to the designed system are done to account for this. Consequently, we evaluate the practicality and security related to these changes.

Through evaluation and analysis, we find that to minimize the expenses of a data owner that utilize our proposed system, the number of peers should be restricted to one in each path. In other words, one peer to deliver a PoS challenge to the data storer. We show that this also improves the fairness of remuneration payout to peers. Subsequently, we analyse how security is affected by always utilizing one peer in each path. We discover that this improves the prevention of the most critical attack, that is drop attack, while it weakens the prevention of a less critical attack, that is release-ahead attack. Through these analyses, we recognize that the benefits greatly outweigh the drawback, and we make a suggestion that data owners should select exactly one peer per path in their services.

We show that the smart contract that implements the design is thoroughly tested by creating a test setup for experiments. This test setup consists of a configuration of a private, local Ethereum blockchain instance. In this blockchain, we test relevant functionalities that our design depends on. That includes the module for P2P communication through the Whisper protocol and various off-chain behaviour. The smart contract is deployed in this network, and we test each function in different service configurations to obtain gas costs.

1.1 Use Cases

There is currently substantial ongoing research on decentralized storage and distribution solutions, and it is increasingly prevalent in terms of popularity and investments [4, 6, 7]. As with their counterpart, centralized storage solutions, these aim for fault-tolerance, no downtime and resistance from denial-of-service attacks. However, the benefit of decentralized storage solutions is that they do not possess the inherent weaknesses of trust-based models which are employed in current centralized storage solutions. Today, most of the storage architectures rely on trusting a service provider, such as Microsoft and Google, on storing and transferring data. In a decentralized storage solution, the stored data is encrypted and spread out to several independent nodes to both provide privacy and redundancy. Additionally, PoS is used to maintain data integrity in order to eliminate the need for trust between storage providers and data owners.

Decentralized storage systems are complex and comprehensive systems, and one of the most important parts of such systems is PoS. The inspiration for the goal in this paper is derived from the research of decentralized storage solutions. The contribution done in this work offers a mechanism that can be integrated into a decentralized storage system issuing self-emerging PoS challenges.

Our contribution is not limited to be integrated into decentralized storage systems. It should also be possible to incorporate our proposed system on top of existing storage infrastructures. A realistic and practical use case for PoS with self-emerging challenges is in combination with a cloud storage service. The system can be used to prove the integrity of a client's data in the cloud. The system would issue challenges to the cloud storage service provider, and prove that the data still exists without having to retrieve actual data from the cloud service. Additionally, the agreement between the service provider and client can be set such that the service provider receives its monthly subscription fee only if it correctly answers the challenges of the client.

The implementation of this system is relevant for both individual and industry use. Individuals use cloud storage services for several purposes, e.g. storage of large multimedia files or data backup of devices. Several service providers offer such solutions for individuals, where Dropbox and OneDrive are only a few examples. The solution presented in this thesis is beneficial for individuals since it provides security that the data exists even though they do not access it.

This solution is also beneficiary for companies that store big data off-site. For instance, ample amounts of sensor data are produced every day. Storing the data at a remote location is a common choice. The system that is presented in this paper provides reassurance of safe storage to companies when they outsource the job of storing.

1.2 Outline

The remaining chapters of this thesis are structured as follows:

Chapter 2 presents relevant background material about PoS and the Ethereum platform. It also provides a summary of the timed-release protocol in [5].

Chapter 3 describes the design choices of the system providing self-emerging challenges using smart contracts. This chapter will highlight our contributions by detailing the modifications and extensions done relative to the timed-release protocol. Furthermore, this chapter provides a discussion of interesting alternative designs.

Chapter 4 details the test setup used to experiment with relevant parts of the implemented system. The test setup includes a private, local Ethereum blockchain deployed in a Docker environment to experiment with the Whisper protocol and off-chain behaviours.

Chapter 5 discusses and evaluates the design based on experiments and analysis. The evaluation is done based on Ethereum gas costs, remuneration to peers and security. Lastly, in this chapter, thoughts on future directions are provided.

Chapter 6 concludes this thesis.

Chapter 2

Background

This chapter provides the relevant background material that is used for the design of the proposed system. We will in the following present an introduction to PoS and the Ethereum platform. Furthermore, we provide a summary of the timed-release service protocol in [5].

2.1 Proof-of-Storage

Proof of data integrity by retrieval is a costly operation since this has a negative impact on both hard drive I/O and network bandwidth. Proof-of-Storage (PoS) schemes allow a verifier V to send data to a prover P and verify that the integrity of its data is maintained without the need for retrieval of the entire data [4]. An example of V and P are a client and a storage provider, respectively. In some PoS schemes, V is not required to be the data owner. That is, a client may outsource the role of V to a third-party. PoS schemes are used in cloud storage and decentralized storage networks since clients in both cases outsource the responsibility of storing data and need to ensure that the integrity of the data is maintained [8]. Common in PoS schemes is that V calculates and stores a set of probabilistic challenges and corresponding proofs while it still possesses its data. Then, V may issue challenges to P. Upon receiving a challenge, P proves that the integrity of the data by correctly responding to the challenge. P is marked faulty if it does not respond with a valid proof or fails to respond to a challenge. An example of such a scheme is the Smash-Proof in the Swarm P2P storage [6].

2.1.1 Public Verification

Public verification is a property within the context of PoS [8]. A PoS scheme is publicly verifiable if V is a party that can verify the integrity of data without possessing private data. That is, a scheme that possesses the publicly verifiable property allows a third-party to verify the integrity of data based on public information. For example, a PoS scheme with this property may allow third-parties to act as V by using the public key of the data owner [9]. Without public verifiability, only parties that possess private information may act as V. Hence, it is possible to prove data integrity and availability, but it does not allow nodes to gain of reputation from verifying. Schemes that do not offer public verification are useful in cloud storage systems.

The publicly verifiable property is useful in the context of decentralized storage networks, such as Filecoin [7]. In Filecoin, storage nodes submit their proofs publicly to the blockchain, and any node in the network can verify these proofs without the need of access to the original data. Since the proofs are stored on the blockchain, they can be verified at any time.

2.2 Ethereum

Ethereum is an open-source blockchain platform that uses a Proof-of-Work consensus algorithm [10]. Like most other blockchain platforms, the functionalities offered revolves around its token. This token is called Ether, and it is used for transactions between accounts and to reward mining nodes for the computational power that they provide to the network. Unlike several other blockchain platforms, Ethereum provides the possibility to program and host decentralized applications within the blockchain. This allows developers that wish to create decentralized applications using blockchain technology to do so without having to implement the underlying mechanics of a blockchain.

Ethereum has a variety of built-in services. One of these is the Whisper protocol that is extensively utilized in our proposed system. It is a P2P communication protocol for decentralized applications as well as nodes in the same Ethereum network. It is designed for small data transfer and security against traffic analysis. Security against traffic analysis is offered through total darkness, meaning that the data that is to be transferred is sent to every listening node, but only the intended recipient can decrypt the data. The drawback of this protocol is the unpredictable latency, which consequently means that communication does not happen in real time.

2.2.1 Smart Contracts

Smart contracts are user-defined digital protocols for the execution of transactions. Since smart contracts are run by miners in the Ethereum network, they are resistant to downtime and interference from third-parties. The smart contract system in Ethereum implements a Turing complete language. This makes the Ethereum platform a preferred environment for the development of smart contracts. The applications of smart contracts are relevant in many sectors. These can be from voting polls in governments to health records in the health care sections. The possibilities are endless since any traditional contract can be ported to a smart contract.

The execution of smart contracts is done in the Ethereum Virtual Machine (EVM). While there exist languages for defining a smart contract, the code compiles to a set of instructions, called opcodes. The opcodes are again encoded to bytecode, which is what the EVM can interpret. Since the numerous unique opcodes combined results in Turing-completeness, this means that the EVM is able to compute most tasks with enough resources.

Smart contracts define a set of functions that peers in the blockchain can invoke to interact with them. Interactions which include operations that modify the state of the contract cost gas. Examples of interactions are transactions of data or currency. In contrast, all read operations are free.

2.2.2 Gas

Gas is what drives Ethereum. It is a unit of cost that represents the computational effort needed to execute a variety of operations [10]. Besides acting as a mechanism to make the execution of denial-of-service attacks infeasible, gas acts as compensation to miners that provide their computational power. A transaction includes gas price and gas limit. The gas price is a value of Ether that is paid per gas unit for the computation costs that arises from executing the transaction. The gas price for a transaction is chosen by its executor. Since the computational effort of execution of the transaction is provided by miners in the network, the miners decide to execute transactions that yield higher compensations, i.e. gas price. Therefore it is essential that the gas price is set high enough so that the transaction is picked up by miners. For this, there exists functionality in Ethereum, where the network suggests a gas price based on recent gas prices in the network. The amount of required gas for the execution of a transaction cannot always be predicted. Therefore, a gas limit is determined by the client, which is the maximum amount of gas the client is willing to pay for. This mechanism is implemented so that operations are finite and costs approximately predictable. The gas limit is paid before the transaction takes place, and cannot be changed at a later point in time.

In distributed systems, the performance of an application is often measured in throughput and latency. Although measurements of such properties of a smart contract may yield meaningful results, in the case of decentralized applications built on blockchain technology, these properties are mainly controlled by the blockchain platform. The smart contract is run in the EVM. Therefore, the underlying technology of Ethereum restricts the performance of smart contracts. Both latency and throughput are dependent on block time and by how fast miners select and execute transactions. Block time in the Ethereum network is typically between 10-20 seconds [11], and miners select to mine the transactions with the highest gas prices. Therefore, the performance, measured in terms of throughput and latency, of an application that utilizes smart contracts is often rendered by the properties of the underlying blockchain platform.

2.3 Timed-Release Protocol

The paper [5] proposes a timed-release service protocol for self-emerging data using smart contracts in Ethereum networks. The proposed protocol allows any pairs of data sender and receiver to set up a service for timed-release data and employs Ethereum peers to partake in the system formed by the smart contract by offering remuneration for their service. The remuneration and compensation for invoking smart contract functions are paid by the sender and receiver of the data.

Protection against various types of attacks that are relevant for the timed-release protocol is shown by modelling the protocol as an extensive-form game with imperfect information [12]. Extensive-form game is used to model and analyze the participants' possible strategies. The attacks that are discussed are post-facto attacks, drop attacks and release-ahead attacks. Post-facto attacks are attacks where peers target to partake in and obstruct the process of a known data sender. This type of attack is avoided since it is assumed that peers register to partake in the protocol before the data sender has announced its participation in the protocol. Drop attacks are any attack where the data fails to reach the recipient. In release-ahead attacks, the data is released before the intended release time. Through modelling the system as an extensive-form game, both drop attacks and release-ahead attacks are shown to be the least rewarding strategy of a participant in the protocol when rewards and sanctions are used to prevent such behaviours.

There are several use cases of the timed-release protocol. An example is secure auction systems where it is crucial that bidding information is kept secret from the auction participants until all bids arrive. A similar application is secure voting mechanisms where votes should not be accessed until the end of the polling process, since revealing the votes could affect the result of the poll. Another use case is copyrights-aware data publishing where data can automatically be released when copyright expires.

The timed-release service protocol involves three different participants to enable self-emerging data. The following describes the roles of the participants in the timed-release protocol.
Data sender (S): The private data is encrypted using a secret cryptographic key and sent to a cloud storage service by S. Furthermore, S sends the encrypted secret key into the blockchain infrastructure, which is only released to R at the release time.
Data recipient (R): The encrypted data is available to R at any point in time. However, the data can only be decrypted when it has received the secret key at the release time, which is determined by S.
Peer (P): The secret key is routed through multiple Ps. The role of the P is to store the encrypted secret key for a certain amount of time, determined by its working window and S, and route it to the next participant.

The timed-release protocol consists of four protocol components. The following presents a summary of the various components.
Peer registration: A new P can register at any time by paying a security deposit to the smart contract. It will then be added into a pool of Ps. After registration, the properties, including the address and working window, of every P is public knowledge to the network.
Service setup: This component serves the purpose of allowing any pair of S and R to register and establish a timed-release service by paying remuneration up front and submitting the selected Ps from the common pool of Ps.

Service enforcement: After the set up of the service, every participant needs to follow the ground rules of this protocol component in order to successfully release the data at the correct time. Any malicious behaviour results in confiscated deposits of the faulty participants.

Reporting mechanism: This component offers functionality to report malicious behaviour by the innocent participants. To be able to detect every misbehaviour, the protocol relies on rewarding participants that report such incidents.

Chapter 3

Design

In this chapter, we describe the design of the decentralized system for PoS using self-emerging challenges built on blockchain technology. Although the system is implemented to utilize Ethereum, its design is applicable for any blockchain platform that supports smart contracts and transaction of currency. First, we present an overview of the architecture. Then, we will look at the different components of the system, namely the smart contract components and the various local behaviours and modules of the participants in the system.

We focus on the design of a scheme of a decentralized system for PoS using self-emerging challenges. We assume that the data storers have a storage service outside of our proposed system and that the data owners pay a fee for the services of their respective data storers through an already established agreement. With these assumptions, we focus the design on the PoS aspect and remove payments between the data owner and data storer in the design. Although, incorporating the payment of data owners to data storers is uncomplicated by utilizing smart contracts, not including this is a calculated choice since it allows data storers to partake in the system without the need to adapt their already implemented and well-established infrastructures for storage and payments.

The design of our proposed system is based on the timed-release protocol in [5] since we use its idea of self-emerging data using smart contracts for PoS challenges. Therefore, there are several similarities. Although the contents of the protocol components differ, we have kept the names of the protocol components since they correctly describe their purpose. Also, the participants can be compared to each other. Peers have the same name and role in both the timed-release protocol and our proposed system. However, the data sender can be compared to the data owner, and the data receiver can be compared to the data storer. The names of these two participants are changed since their roles serve different purposes and have different behaviours in our system. Furthermore, the design of each protocol component and algorithms in our proposed system and the timed-release protocol differ to varying degrees. To provide a comprehensive and coherent presentation of our proposed system, we choose to present it from the bottom up, and at the end of each relevant section emphasize the differences between our system and the timed-release protocol.

3.1 Design Overview

The design of this system is heavily based on the use of a smart contract, which is a trusted intermediary between the participants that are involved. It acts as a hub for public communication. This includes the organization of the peers that wish to participate in the process, submission of challenges, and submission of answers to the challenges. In addition to the smart contract, the design includes operations that are done locally by various participants and also secure P2P communication. Both the design of the smart contract and the different local behaviours of the involved participants will be described in the following sections. We will first present the design overview of the system and then proceed to describe each component in detail.

The system consists of three types of entities. These are data owner, peers and data storer. PoS challenges are sent from the data owner and are kept secret from the data storer for fixed periods of time by a set of peers in a path. The number of peers in a path depends on the available peers and their available working periods, which is used by the peer selection algorithm that will be discussed in Section 3.4.2. In addition, the total amount of paths are not set by the system but are rather decided by the number of challenges that the data owner wishes to issue. Each path consisting of peers represents a unique PoS challenge that is to be delivered to the data storer at a specific time decided by the data owner. Upon reception of a challenge, the data storer has to calculate an answer and deliver it to the smart contract within a deadline. If it either does not answer a challenge correctly or does not to meet the deadline, it will fail to prove the integrity of the data. Most of the interactions by the participants are done with the smart contract; however, the challenges are sent through a path consisting of peers by the utilization of the P2P protocol, Whisper, to limit the gas cost required from the execution of transactions.

The structure of the system with the different entities is illustrated in Figure 3.1. It visualizes a scenario where three self-emerging challenges are issued from the data owner. In this case, the data owner has chosen that each challenge is to be delivered to the data storer with a constant time interval t apart from the service setup time.

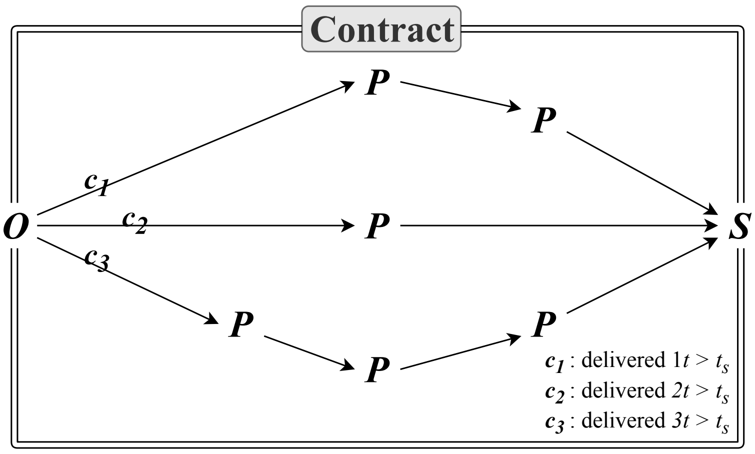
Contract
P
P
P
P
P
P
O
S
c_1
c_2
c_3
c_1 : delivered $1t > t_s$
c_2 : delivered $2t > t_s$
c_3 : delivered $3t > t_s$

Figure 3.1: The general architecture of the system. This particular setup consists of data owner O, data storer S and a different amount of peers P in each path. Each path is used to send a challenge c, where each c has a particular release time.

3.2 Roles in the System

This section will provide an introduction to the roles of the different participants in the system by describing their motivation and off-chain behaviour. To provide a brief overview of the roles, some technical descriptions are omitted but will be discussed in further detail in the following sections.

3.2.1 Data Owner

The data owner is a participant that has the interest of receiving proof of data integrity for their remotely stored data. The data owner employs our system to carry out a set of PoS challenges to ensure that data integrity is maintained, assuming that its data storer offers to participate in our proposed system. Since we assume that this participant pays for the storage service of its data storer in another system, it is interested in keeping the expenses by employing this system at a minimum. The following describes the behaviour of the data owner and its interaction with the other participants.

The data owner calculates and stores a set of PoS challenges and answers using the PoS algorithm, which will be described in Section 3.4.3, prior to only relying on the remote storage service of the data storer. Furthermore, it registers to the system by signing the smart contract. In order to set up the service, it has to run the peer selection and remuneration calculation algorithms that are presented in Sections 3.4.1 and 3.4.2. When its data storer also has agreed on participating in the system, it is allowed to set up the service by submitting a set of required data. That is, the selected peers for its service, pay remuneration to selected peers upfront, and pay a deposit for prevention from misbehaviour. After the setup and before the working window, i.e. service period, of the first peer in a path, the data owner calculates a set of certificates for all peers and the data storer. As described in Section 3.3.3, these certificates are used for a security mechanism and are concatenated and encrypted using onion encryption. Furthermore, it submits the hashes of the certificates, the hashes of the answers to the challenges, and Whisper keys encrypted with the public key of the next peer in the paths. A Whisper key is a symmetric encryption key that is used to send challenges directly to the next participant in a path using the P2P Whisper protocol. This behaviour is detailed in Section 3.4.4. After transferring the challenge to the first peer in a path, the protocol does not require any further actions from the data owner for that particular path. The data owner may go offline after transferring the challenges into all of its paths.

3.2.2 Data Storer

Like the data owner, the data storer has the interest of participating in the proposed system to prove data integrity. Since the system does not require any payments from the data storer, there will be no expenses related to participating in this system except computing power and some gas cost related to invoking smart contract functions. However, given that the data storer is one of multiple data storers in the system, it will gain trust and a positive reputation among clients that it correctly and safely stores data if it manages to correctly answer the challenges of the data owners. Therefore, the computing power and gas cost can be compensated for with a positive reputation and consequently, new clients. The following describes the behaviour of the data storer and the interaction with other participants in the system.

The data storer signs the smart contract to form a pair with the data owner. It then waits until a Whisper key encrypted with its public key is submitted by the last peer in a path. Upon submission of the Whisper key, the data storer decrypts this key with its private key and starts to listen to the network for the incoming challenge. After reception of the challenge, it disjoins the certificate that is concatenated to the challenge and submits the certificate to the smart contract. If the verification of the certificate is successful, it verifies the behaviour of itself and the previous peer. Furthermore, it calculates the answer to the challenge using the PoS algorithm that is presented in Section 3.4.3, and submits it to the smart contract. If the smart contract verifies that the answer is correct, the data storer has successfully proved data possession and integrity at the release time of the PoS challenge. The data storer needs to repeat this process for all paths.

3.2.3 Peers

The peer is a participant that offers its service for payment. Unlike the data owner and storer, its primary interest is not proving data integrity, but rather receiving a return on its deposit. The following describes the behaviour of the peer and the interaction with other participants in the system.

The peer announces its interest in participating in services by registering to the smart contract. Since the data storer has not set up the service at this point in time, the peer has no knowledge of the identity of neither the data owner nor storer. It registers to the system by paying a desired deposit and submitting its public key and working window. If the peer does not get selected for any service, it can change all of its attributes at any time. After it has been selected as a peer in a service, it can go offline until the start of its working window. At the beginning of its working window, it waits until a Whisper key is submitted from the previous participant in the path. Upon reception of the Whisper key, the peer decrypts this key with its private key and starts to listen to the network for the incoming challenge. After reception of the challenge, it disjoins the certificate that is concatenated to the challenge and submits the certificate to the smart contract. It also submits a new encrypted Whisper key for communication with the next participant in the path. If the verification of the certificate is successful and it has submitted the Whisper key successfully, it verifies the behaviour of itself and the previous peer. Lastly, using the Whisper key created for communication with the next participant, it transfers the decrypted challenge to the next participant before its working window is over. After a successful service, it receives remuneration for its service, and it can either withdraw its deposit or keep it in the smart contract for a new service.

3.3 Smart Contract

The smart contract implemented in this system builds upon the protocols that are presented in [5]. However, since the implementation of the protocols was not included in the paper, we have built a smart contract from the ground up. The following subsections describe the various protocols that make up our proposed smart contract. Each protocol consists of several functions. Table 3.1 shows the most significant functions in each protocol with their respective callers. Furthermore, we assume that the native Ethereum environment variables are accessible for every function. These include the address of a function invoker, known through the variable *msg.sender*, and the payment to a function, if any, known through *msg.value*. We also assume that function modifiers allow invocations of functions only accessible by participants that exist in the system. These modifiers will be marked as a comment in the algorithms next to the function declarations.

Protocol	Function	Caller
Register	newPeer	Peer
	updatePubKey	Peer
	updateBalance	Peer
	updateWindow	Peer
Setup	ownerSign	Owner
	storerSign	Storer
	setup	Owner
Enforce	setProps	Owner
	verifyCert	Peer, Storer
	setWhisperKey	Peer
	verification	Peer
	answer	Storer
Report	releaseReport	All
	dropReport	Peer, Storer

Table 3.1: The main functions of the smart contract and their callers.

3.3.1 Peer Registration Protocol

The peer registration protocol allows nodes in the Ethereum blockchain, commonly referred to as peers, to participate in the system. Algorithm 3.1 details the functions with their respective pseudo code. Peers register through the `newPeer` function by submitting a set of attributes. These are public key *pubKey* and working window $window = [t_b, t_e]$ that is their offered service period. In addition, peers need to pay a deposit d^s through this function. This deposit is used to prevent misbehaviour. The smart contract adds and initializes other attributes to the peer that are used in other protocol components. These are $hasFrozenDeposit$ to determine if the peer is part of a service and consequently has its deposit frozen, the hash of its certificate s_h and its Whisper key k_w. Furthermore, a set of update functions are provided for a peer that wishes to change its respective attributes after registration, namely `updateBalance`, `updateWindow` and `updatePubKey`. While the algorithms for updating working window and updating public key are self-explanatory, the function updating balance is used to both add more funds to a peers deposit and to withdraw the entire deposit in its account by passing zero to the function.

When registering to the smart contract, the peer is added to a pool of peers. Data owners use this pool to choose which peers to include in their process based on the peers' working windows and deposits, which are used by the system for penalizing misbehaviour. Since a challenge needs to be stored in the blockchain network for a certain amount of time before it is delivered to the data storer, the working windows of peers are used to decide which peers that are chosen to participate in a path. A path in which a challenge is routed can be divided into multiple peers if no peer in the pool can service the period as requested by the data owner. The system has been designed such that a peer cannot participate in multiple paths or processes. Although this could have been implemented, it has been a calculated choice to not including this in the design for the sake of simplicity. Section 3.5 discusses what accommodations that are needed in order to implement this feature.

A high-level overview of the system is shown in Figure 3.2, where several data owners may use the pool of peers to form paths in order to send their challenges to their respective storers.

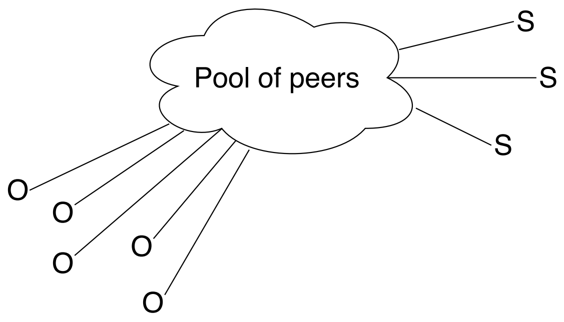

Figure 3.2: A high-level overview of the smart contract. Several data owners O use the pool of peers to form paths for their services. Through their paths, they send challenges to their respective data storers S.

Algorithm 3.1 Peer Registration Protocol

```
$\hat{p} \leftarrow \emptyset$;                                    ▷ Initialize set of peers.

function NEWPEER($pubKey, window$)
    if peerExists($msg.sender$) then
        revert("Peer is already registered.");
    end if
    $\hat{p}[msg.sender] \leftarrow \{d^s : msg.value,$
                         $hasFrozenDeposit : false,$
                         $k_{pub} : pubKey,$
                         $s_h : \bot,$
                         $certVerified : false,$
                         $passedVerification : false,$
                         $k_w : \bot,$
                         $T^w : window\}$;
end function

function UPDATEBALANCE()                                 ▷ Caller is a registered peer.
    $require(\neg p[msg.sender].hasFrozenDeposit$, "Peer is partaking in an ongoing process.");
    if $msg.value = 0$ then                      ▷ Allows peer to withdraw its deposit.
        $msg.sender.transfer(p[msg.sender].d^s)$;
        $p[msg.sender].d^s \leftarrow 0$;
    else
        $\hat{p}[msg.sender].d^s \leftarrow \hat{p}[msg.sender].d^s + msg.value$;
    end if
end function

function UPDATEWINDOW($window$)                          ▷ Caller is a registered peer.
    $require(\neg p[msg.sender].hasFrozenDeposit$, "Peer is partaking in an ongoing process.");
    if $window.t_b < now \lor window.t_e < now \lor window.t_e < window.t_b$ then
        revert("Start and end times must be later than the current time, and end time after start.");
    end if
    $\hat{p}[msg.sender].T^w \leftarrow window$;
end function

function UPDATEPUBKEY($pubKey$)                          ▷ Caller is a registered peer.
    $require(\neg p[msg.sender].hasFrozenDeposit$, "Peer is partaking in an ongoing process.");
    $\hat{p}[msg.sender].k_{pub} \leftarrow pubKey$;
end function
```

This protocol differs little from the corresponding peer registration component in the timed-release protocol. In [5], an overview description is given for the peer registration process; however, no implementation or pseudo code is provided. Therefore, Algorithm 3.1 and its data structures are built from the bottom up for our proposed system. This is also the case for all protocols presented in Section 3.3.

3.3.2 Service Setup Protocol

The service setup protocol allows data owners and storers to register in the system. When both are registered correctly and have become a pair, the data owner may set up a service with one or more challenges. For this, the smart contract requires the data owner to submit selected peers for its paths, pay the rewards and gas compensations of peers up front, and pay a deposit that is the same deposit required by peers multiplied with the number of paths. As discussed in Section 3.3.4, the data owner, like the other participants in the system, may execute attacks. Therefore, the system is designed such that it requires a deposit from the data owner for each challenge to prevent misbehaviour. The smart contract calculates if the payment is sufficient based on the number of selected peers and their working windows. Furthermore, it checks if the selected peers fulfill the requirements, that is, that they have sufficient deposits and that their deposits are unfrozen. If the payment of the data owner is correct, and the selected peers fulfill the requirements, the smart contract freezes the peers' entire deposits and accepts the payment of the data owner. The remuneration calculation and the peer selection algorithm are presented and discussed in detail in Sections 3.4.1 and 3.4.2. Peers' deposits are frozen upon service setup to insure that peers follow the protocol correctly.

Algorithm 3.2 shows the functions that make up the service enforcement protocol. The data owner signs up to the system by using the function `ownerSign`. The function inputs the address of the data owner's storer and marks the owner as signed up to the system. The data storer signs up to the system by invoking the `storerSign` function. This function checks if the address of the storer is the same that the data owner registered. It also lets the data storer submit its public key *pubKey* that is used in the service enforcement protocol. The data owner sets up the system using the `setup` function that inputs *paths* which is a set of selected peers for each path, the integer *remuneration* for how much the data owner has paid for remuneration to all peers in all paths, and the integer *deposit* for the deposit that is required from the selected peers. This function requires that both the data owner and data storer have signed up to the system beforehand. It further checks that the payment by the data storer is equal or greater than the sum of remuneration and deposit multiplied with the number of paths. If this yields true, it calculates and checks if the remuneration paid by the data owner is sufficient based on the working windows of the peers in all paths. It also checks that the selected peers have sufficient deposits as required by the data owner and that they do not partake in other processes, i.e. that their deposit account is unfrozen. Additionally, it checks that the submitted peers contain no other addresses than the registered peers in the system.

These operations are done using the helper functions `isOwnerPaymentSufficient` and `peersMeetReq`. If the conditions are met, the smart contract accepts the setup by freezing the deposits of the selected peers.

Algorithm 3.2 Service Setup Protocol

```
1: $owner \leftarrow \perp$;
2: $storer \leftarrow \perp$;
3: $\hat{r} \leftarrow \perp$;
4: $\tilde{P} \leftarrow \emptyset$;    ▷ Initialize set of paths. $\tilde{P}[i]$ represents path. $\tilde{P}[i][j]$ represents peer in path.
5: $d^s \leftarrow \perp$;
6: $terminated \leftarrow \emptyset$;    ▷ Initialize set for logging if a path is terminated.

7: function OWNERSIGN($storerAddr$)
8:     $owner \leftarrow \{addr : msg.sender,$
                  $\widehat{k_w} : \emptyset$
                  $hasSigned : true\}$;
9:     $storer.addr \leftarrow storerAddr$;
10: end function

11: function STORERSIGN($pubKey$)
12:     if $msg.sender = storer.addr$ then
13:         $storer \leftarrow \{addr : msg.sender,$
                        $k_{pub} : pubKey,$
                        $\widehat{s_h} : \emptyset,$
                        $hasSigned : true$
                        $certsVerified : \emptyset;$
                        $hasProved : \emptyset;$
14:     end if
15: end function

16: function SETUP($paths, remuneration, deposit$)    ▷ Caller is owner.
17:     $require(owner.hasSigned \land storer.hasSigned,$ "Both owner and storer must sign.");
18:     $require(msg.value \geq remuneration + deposit \times count(paths),$
            "Payment is insufficient relative to inputted values.");
19:     $\tilde{P} \leftarrow paths$;
20:     $d^s \leftarrow deposit$;
21:     $\hat{r} \leftarrow remuneration$;
22:     $terminated \leftarrow [false] \times count(paths)$;    ▷ Mark all paths as not terminated.
23:     if $\neg isOwnerPaymentSufficient(\tilde{P}, \hat{r}) \lor \neg peersMeetReq(\tilde{P}, d^s)$ then
24:         revert("Setup request does not fulfill requirements.");
25:     end if
26:     $freezeDeposits(\tilde{P})$;    ▷ Freezes the deposits of each selected peer in all paths.
27: end function
```

As with the peer registration protocol, Algorithm 3.2 is based on the timed-release protocol. However, the functionality presented for the service setup in our proposed system differs to a greater extent from the timed-release protocol. The design choice of pairing the data owner to the data storer is made independently for this system as the timed-release protocol does not detail any description, nor present an obvious explanation on how pairing is done. Furthermore, since this system allows issuing multiple challenges, the setup process requires submission of selected peers for several paths. This involves the creation and management of more complex data structures. An example of this is the more extensive remuneration calculation, as it is different for varying timelines of challenges and consequently has to be done for each path. Moreover, since the roles and motivations of the participants in this system differ significantly from the participants in the timed-release protocol, this system is designed such that the receiver of the self-emerging data, the data storer, does not pay any deposit, nor receives any remuneration.

3.3.3 Service Enforcement Protocol

The service enforcement protocol provides a set of functions that are needed in order to establish communication between participants in a path, as well as functions that are needed by the participants to verify their behaviour. Successful delivery of a self-emerged challenge can be achieved if the involved participants in the process follow this protocol correctly.

The challenge that is sent through a path is onion encrypted with the public keys of the peers and the data storer in reversed order. Figure 3.3 illustrates the encryption scheme for a path consisting of three peers. To decrypt the challenge, the first peer has to decrypt a layer, then second peer, then third peer and lastly the data storer. With this encryption scheme, the order of reception is enforced. In our design, the certificates are sequences of fixed-length, random bits created by the data owner. They are introduced to let participants prove that they have successfully received and decrypted a layer of the routed challenge. The certificates are appended to the challenge for each public key encryption layer, which consequently makes them accessible by decryption by the intended participant using their private key. The challenge and certificates are sent between participants in a path using the Whisper protocol. In order to create a channel between two participants, the sender should create a symmetrical Whisper key, encrypt it using the public key of the receiver, and submit it to the contract. Subsequently, the receiver may download the encrypted Whisper key from the smart contract, decrypt it using its private key, and use it to listen to the Whisper channel. Figure 3.1 shows an example of Whisper channels, represented by arrows, that are established between participants in paths in order to route the challenges to the data storer.

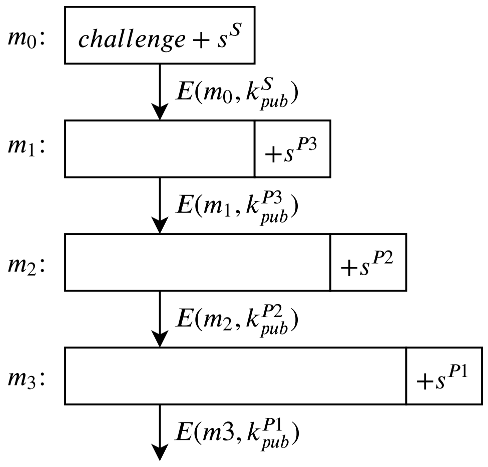

Figure 3.3: Onion encrypted challenge for a path consisting of peers $P1 - P3$ and data storer S. A unique certificate s is appended to each message m and encrypted with the public key k_{pub} of the intended receiver for each layer.

Subsequent to executing the service setup protocol, the service enforcement protocol lets the data owner submit the necessary data to the smart contract. These are hashes of the certificates, hashes of the answers and challenges, and its Whisper key encrypted by the public key of the next participant in the path. Furthermore, the data owner needs to send the onion encrypted challenge with the certificates to the next peer in the path using the Whisper protocol. All of these operations have to be done before the working window of the next peer in the path. Before its working window is finished, each peer has to decrypt a layer of the received data in the Whisper protocol module, submit the obtained certificate to the smart contract, submit the encrypted Whisper key, and verify its own behaviour and consequently the behaviour of the previous participants. Lastly, it needs to transfer the encrypted challenge to the next participant in the path using the Whisper protocol module. Upon reception of the encrypted challenge and certificate, the data storer has to decrypt the last layer, submit the obtained certificate to the smart contract, verify its own behaviour and consequently the behaviour of the previous participants in the path by answering correctly to the received challenge. If the verification done by all peers and the data storer passes, the contract unfreezes the deposits of the peers and pays remuneration to every peer that has finished their job correctly. However, if the verification done by peers or data storer does not pass, the smart contract terminates the process in that path and marks the last participant that failed to pass the verification as guilty. Furthermore, the smart contract pays remuneration to the innocent peers, unfreezes the innocent peers' deposits, and pays the confiscated deposit and unused remuneration to the data owner.

Multiple invocations of the `verification` function in a path are done to be able to unfreeze the deposits of innocent peers as soon as possible. In [5], it is shown that any rational participant in this system will always choose to submit the Whisper key and certificate within its working window. Since their protocol is designed as an extensive-form game with imperfect information, by utilizing game theory, they prove that the most rewarding strategy of any participant in the system is to correctly follow protocol without misbehaving. Their proof is based on reaching the Nash equilibrium [13]. Their proof is applicable for peers in our system as well since we use the same penalize and reward scheme. However, since our system does not require the data storer to pay for partaking in the system, we cannot entirely base our system on this analysis. We discuss this issue in Section 5.4.

Algorithm 3.3 lists the functions that make up the service enforcement protocol. The data owner submits the hashes of the certificates for each participant in all paths, the hashes of the certificates of the data storer for each challenge, the hashes of all challenges, the hashes of corresponding answers to the challenges and its Whisper key by inputting $certHashes$, $storerCertHashes$, $challengeHashes$, $answerHashes$ and $whisperKey$ to the `setProps` function. The function adds the $certHashes$ to all selected peers in each path, $storerCertHashes$ to the data storer and the Whisper key to the data owner. It furthermore registers both hashes of the challenges and answers. Peers and the data storer verify their obtained certificate by submitting the plain text certificate $cert$ and the path that it is a part of, represented as an integer, using the function `verifyCert`. This function calculates the hash of the submitted certificate and compares it to the hash of the certificate submitted by the data owner. If the hashes are equal, it registers that the peer has correctly verified the certificate. Peers submit their Whisper keys using the function `setWhisperKey`. The verification process by a peer is done by invoking the `verification` function after submission of the certificate and the encrypted Whisper key. The function inputs an integer $path$ which represents the path of which the invoker is a part. If both of these are submitted and if the path is not already terminated, the verification will pass. However, if these are not submitted within the working window of the peer, it will not pass, and a drop attack is detected. Failure of verification and consequently termination of the service in a path is handled by unfreezing the deposits of innocent peers and data owner, and paying the remuneration to innocent peers using the utility functions `unfreezeDeposits` and `remunerationPayout` respectively. The confiscated deposit and remuneration are then transferred to the data owner. If a participant has followed the ground rules of the system honestly but still cannot verify its certificate correctly, this means that a drop attack has been executed by another participant. Therefore, the participant may report the drop attack, as described in the reporting protocol in 3.3.4, without invoking `verification`. The verification process by the data storer is done by invoking the `answer` function after submission of its certificate. This function inputs the integer variable *path* that represents the path of which the challenge is a part, and the plain text challenge answer $answer$. The function calculates the hash of $answer$ and compares it to the hash of the answer submitted by the data owner a_h. The verification, whether failure or success, is registered by the function. Lastly, the function unfreezes deposits of all peers and data owner and pays out remuneration to all peers in the path using the utility functions.

Algorithm 3.3 Service Enforcement Protocol

```
1: $\widehat{c_h} \leftarrow \emptyset$;    $\triangleright$ Initialize variable for hashes of certificates.
2: $\widehat{a_h} \leftarrow \emptyset$;    $\triangleright$ Initialize variable for hashes of answers.

3: function setProps($certHashes, storerCertHashes, challengeHashes, answerHashes,$
            $whisperKeys$)    $\triangleright$ Caller is owner.
4:     for $i \leftarrow 0, certHashes.length$ do
5:         for $j \leftarrow 0, certHashes[i]$ do
6:             $\hat{p}[\tilde{P}[i][j]].s_h = certHashes[i][j]$;
7:         end for
8:     end for
9:     $storer.\widehat{s_h} \leftarrow storerCertHashes$; $\widehat{c_h} \leftarrow challengeHashes$; $\widehat{a_h} \leftarrow answerHashes$;
10:    $owner.\widehat{k_w} \leftarrow whisperKeys$;
11: end function

12: function verifyCert($path, cert$)    $\triangleright$ Caller is selected peer or storer.
13:     $certHash \leftarrow$ keccak256($cert$);
14:     if $\hat{p}[msg.sender].s_h = certHash$ then
15:         $\hat{p}[msg.sender].certVerified \leftarrow true$;
16:     else if $msg.sender = storer.addr \wedge storer.\hat{s_h}[path] = certHash$ then
17:         $storer.certsVerified[path] \leftarrow true$;
18:     end if
19: end function

20: function setWhisperKey($whisperKey$)    $\triangleright$ Caller is selected peer.
21:     $\hat{p}[msg.sender].k_w \leftarrow whisperKey$;
22: end function

23: function verification($path$)    $\triangleright$ Caller is selected peer.
24:     $require(\neg terminated[path]$, "Path is already terminated");
25:     if $\hat{p}[msg.sender].certVerified \wedge \hat{p}[msg.sender].k_w \neq \perp$ then
26:         $\hat{p}[msg.sender].passedVerification \leftarrow true$;
27:     end if
                    $\triangleright$ Incorrect behavior is detected. Service is terminated.
28:     $unfreezeDeposits(\tilde{P}[path] \setminus msg.sender)$;
29:     $remain \leftarrow remunerationPayout(\tilde{P}[path] \setminus msg.sender)$;
30:     $owner.addr.transfer(2 \times d_s + remain)$;    $\triangleright$ Transfer remaining $\hat{r}$ and $d^s$ to owner.
31:     $terminated[path] \leftarrow true$;
32: end function

33: function answer($path, answer$)    $\triangleright$ Caller is storer.
34:     $require(\neg terminated[path]$, "Path is already terminated.");
35:     if $storer.certsVerified[path] \wedge keccak256(answer) = \widehat{a_h}[path]$ then
36:         $storer.hasProved[path] \leftarrow true$;    $\triangleright$ Proven data integrity for the challenge.
37:     else
38:         $storer.hasProved[path] \leftarrow false$;    $\triangleright$ Failed to proved data integrity for the challenge.
39:     end if
40:     $unfreezeDeposits(\tilde{P}[path])$
41:     $remunerationPayout(\tilde{P}[path])$;
42:     $owner.addr.transfer(d_s)$
43:     $terminated[path] \leftarrow true$;
44: end function
```

Although inspiration is derived from the timed-release protocol, the design choices of Algorithm 3.3 deviate significantly. As the service setup protocol, this protocol depends on and pursue the more complex data structures than the ones that are required for the timed-release protocol since it allows the self-emerging data in multiple paths. Consequently, a management mechanism for multiple paths in which the participants invoke functions for verification is created by requiring participants to submit in which path they are performing their service. Moreover, the design choices differ considering the verification of behaviour done by the receiver of the data, i.e. the data storer. Instead of verifying its behaviour by executing the same functions as the peers, the system is extended so it can verify its behaviour and terminate the process of the self-emerging challenge in a path by responding to the challenge.

3.3.4 Reporting Protocol

The reporting protocol offers a set of functions for handling release-ahead attacks and dispute of drop attacks. A release-ahead attack is a type of attack where a peer chooses to release the challenge that is being routed in the path before the release time set by the data owner. A challenge is routed in a path using multiple Whisper channels. The routing of a challenge is, therefore, independent of the smart contract. Hence, it is hard to detect release-ahead attacks. The reporting protocol addresses this problem by providing a function so that the data storer can report this behaviour. Furthermore, since the service enforcement protocol only detects drop attacks and cannot deduce which peer has executed the drop attack between the two peers where the drop took place, this protocol also offers a function for reporting drop attacks in order to resolve a dispute between two peers.

As discussed, release-ahead attacks are hard to detect since the challenges are transferred between participants in a path through the Whisper protocol, which uses end to end encryption. In the system where several data storers participate, it is useful to enable honest data storers to report a release-ahead attack. The report needs to contain a proof that the challenge is received before its time. For that, the challenge only encrypted by the data storer's public key needs to be submitted to the contract before the actual release time. This proof can then be verified by the smart contract, which compares the hash of the submitted proof with the hash of the challenge submitted by the data owner at the setup time. Upon receiving a valid proof, it is shown that the participant that is located before the reporter has released the challenge too early. By enabling the data storer to report such incidents, every participant in the system is informed of the guilty peer since this will remain as public, permanent information in the blockchain. The data owners that wish to set up new processes then have the possibility of excluding peers that are proven to be dishonest from their processes. The data storers that report such incidents are awarded since the system benefits of honest participants. The system is designed so that a rational participant will not carry out a release ahead attack since it will be penalized. A release ahead attack can be caused by a data storer that bribes a peer to execute the attack on its behalf. Although a release ahead attack in principle can origin from any participant, in this system, the beneficiary from such an attack is the data storer. In this system, the amount of penalization is set to be d_s, and this is used as an award $a = d^s$ to the reporter. This means that the executor of the attack will lose its deposit and potential remuneration, and the reporter will gain the executor's deposit if it reports the incident. Therefore, a malicious data storer must be willing to bribe an amount larger than $d^s + r$ to motivate a peer to execute such an attack on its behalf. We will give a more in-depth evaluation about release-ahead attacks in Section 5.4.

It is stated in [5] that a drop attack can be executed in three different ways. We use Figure 3.4 that shows a path consisting of two peers to explain how a drop attack can be executed. The first peer P_1 can execute a drop attack by not sending the correct challenge to the second peer P_2. P_2 can also execute a drop attack by denying that it has received the challenge from P_1. Lastly, it is also considered a drop attack if the data owner O submits fake hashes of certificates to the contract. In all of these types of drop attacks, a dispute between the participants will arise. The mechanism in solving this dispute depends on P_2 reporting the incident, and it assumes that $d^s > v$ and $d^s > v + a$, where v is the value of the challenge and a is the award to the reporter. Upon reporting a drop attack, the smart contract confiscates the deposits of the three participants and awards P_2. It is shown by the game induced by the protocol that this effectively prevents drop attacks for rational players. By reporting the incident, P_2 will lose $d^s - a$, but it will lose d^s if it does not report it. Moreover, in the case where P_1 has followed the protocol correctly, the P_2 will lose $d^s - a$ upon falsely reporting a drop attack. With this logic, the second peer will never launch a drop attack. Furthermore, since the data owner and the first peer will lose their entire deposit upon a report of a drop attack, their rational behaviour is never to launch a drop attack. Following the protocol honestly will result in the highest monetary outcome for all participants. Thus, the Nash equilibrium is reached in the game where all participants are rational. It is important to notice that this does not apply in a scenario where the data storer executes a drop attack by denying that

$$\begin{tikzcd} O \arrow[r] & P_1 \arrow[r] & P_2 \arrow[r] & S \end{tikzcd}$$

Figure 3.4: Simple path consisting of two peers.

it has received the challenge since it does not pay any deposit which can be used for penalization. We will further discuss and evaluate this issue in Section 5.4.

In the following example, we illustrate that rational participants will not execute drop attacks. Let $d^s = 6$, $a = d^s/2$ and $v = 2$, where v is the motivation and gain for a peer to execute such an attack. If P_1 executes a drop attack by not sending the correct challenge to P_2, it will lose its entire deposit $d^s = 6$ and only gain $v = 2$. P_2 will lose $d^s - a = 3$ if it does report the incident, but it will lose $d^s = 6$ if it abstains from reporting. Therefore the rational behaviour of P_1 is never to launch a drop attack, and the rational behaviour of P_2 is to report the incident if P_1 has executed a drop attack. The same logic is also applicable in a case where O executes a drop attack by submitting incorrect hashes of certificates to the smart contract. The motivation of O for executing this attack is, however, not obtaining v since it already possesses this, but its motivation can be external, e.g. spoiling the reputation of the data storer. O will lose its entire deposit $d^s = 6$ upon a drop attack, while P_1 will lose $d^s - a = 3$ by reporting the incident. If P_1 abstains from reporting the incident it will however lose its entire deposit $d^s = 6$. If P_2 executes a drop attack by denying the reception of the challenge from P_1 and falsely reports a drop attack, it will lose $d^s - a = 3$. If it does execute a drop attack and does not report it, it will lose $d^s = 6$. However, if it follows the protocol correctly, it will not lose any of its deposit and receive remuneration. In any way, if P_2 executes a drop attack, P_1 and O will lose their entire deposits. Therefore, it is in the common interest for all peers and O that a drop attack is not executed.

Algorithm 3.4 lists two functions for reporting an attack. Function `releaseReport` allows the data storer to report a release-ahead attack by taking input parameters *path* which is an integer representing the path where the attack has taken place, and *challenge* which is the challenge for that path only encrypted with the public key of the data storer. If the hash of the challenge is the same as the one submitted by the data owner and this has been reported before the release time of the challenge, a release-ahead attack is proven. The function will then unfreeze the deposits of innocent peers and the data owner, and pay remuneration to all innocent peers in the path using the utility functions `unfreezeDeposits` and `remunerationPayout`. The innocent peers are all peers except the peer that executed the release-ahead attack, that is, the last peer in the path before the reporter. Lastly, the reporter gains an award equal to the deposit of the guilty peer.

Function `dropReport` allows participants to report a drop attack to solve the dispute of the attack by inputting the path where the attack has taken place. It is required that the data owner is not the caller of the function since it is the first participant in the path. The function checks if the working window of the last peer in the path before the reporter is finished in order to avoid false reports at times where the reporter should not be able to report drop attacks. If the reporter is not the data owner and the reporter exists as a participant in the path, the deposits of all participants in the path except the reporter, the participant before the reporter and the data owner are unfrozen. Moreover, remuneration is paid to all participants except the reporter and the participant before the reporter. Lastly, an award equal to half of a deposit transferred to the reporter. The award is only paid to the reporter if it is not the data storer. Since the data storer does not pay any deposit, it would always gain $d^s/2$ for falsely reporting drop attacks without having anything to lose.

Algorithm 3.4 Reporting Protocol

```
function releaseReport($path, challenge$)    ▷ Challenge only encrypted with the storer's pubKey.
    $require(\neg terminated[path]$, "Path is already terminated.");
    $lastPeer \leftarrow \tilde{P}[path][count(\tilde{P}[path]) - 1]$;    ▷ Address of the last peer in the path.
    $require(\hat{p}[lastPeer].T^w.te > now$, "Proof not delivered before challenge release time.");
    if $keccak256(challenge) = \hat{c_h}[path] \wedge msg.sender = storer.addr$ then
        $unfreezeDeposits(\tilde{P}[path] \setminus lastPeer)$
        $remain \leftarrow remunerationPayout(\tilde{P}[path] \setminus lastPeer)$;
        $msg.sender.transfer(d^s)$;    ▷ Award reporter with confiscated $d^s$ of faulty peer.
        $owner.addr.transfer(remain + d^s)$;    ▷ Transfer remaining $\hat{r}$ to owner and unlock its $d^s$.
        $terminated[path] \leftarrow true$;
    end if
end function

function dropReport($path$)    ▷ Caller is selected peer or storer.
    $require(\neg terminated[path]$, "Path is already terminated.");
    $require(msg.sender \neq owner.addr$, Invalid caller);
    $faulties \leftarrow \emptyset$;    ▷ Initialize set for adding penalized peers.
    if $msg.sender \neq storer.addr$ then    ▷ Penalize peer and peer before reporter.
        $require(\hat{p}[msg.sender].T^w.t_b > now$, "Invalid invocation time.");
        $i \leftarrow indexOf(msg.sender, \tilde{P}[path])$;    ▷ Index of invoker in $\tilde{P}[path]$.
        if $i > 0$ then
            $require(\hat{p}[\tilde{P}[path][i-1]].T^w.t_e < now$, "Invalid invocation time.");
        end if
        $faulties \leftarrow faulties \cup \tilde{P}[path][i-1]$;
        $faulties \leftarrow faulties \cup msg.sender$;
        $msg.sender.transfer(d^s/2)$;    ▷ Award reporter with $a = d_s/2$.
    else    ▷ Penalize last peer in path.
        $lastPeer \leftarrow \tilde{P}[path][count(\tilde{P}[path]) - 1]$;    ▷ Address of the last peer in the path.
        $require(\hat{p}[lastPeer].T^w.te < now$, "Invalid invocation time.");
        $faulties \leftarrow faulties \cup \tilde{P}[path][count(\tilde{P}[path]) - 1]$;
    end if
    $unfreezeDeposits(\tilde{P}[path] \setminus faulties)$
    $remunerationPayout(\tilde{P}[path] \setminus faulties)$;
    $terminated[path] \leftarrow true$;
end function
```

The design of Algorithm 3.4 possess inspiration from the timed release protocol. The design of the reporting protocol differs from its corresponding counterpart in the timed-release protocol in that it offers the same mechanism for managing multiple paths as in Algorithm 3.3. Furthermore, since the description of the timed-release protocol does not include the amount of the award of the reporters, the design choice for the system is set such that an award for correctly reporting a release-ahead attack is the amount of a deposit, and the award for reporting a drop attack is the amount of half a deposit. This is a calculated choice since the awards are large enough to encourage participants to report incidents but small enough to avoid false reports. Another significant difference from the timed-release protocol is that the system does not offer a mechanism for prevention from attacks by the data receiver, i.e. data storer, to the same extent. This is because the system assumes that service payment to the data storer by the data owner is made externally, and therefore, it does not require a deposit from the data storer. As will be discussed in further detail in Section 5.4, this is accounted for since it is assumed that the data storer has an external motivation to behave honestly. Moreover, another security issue is raised from the design difference of not including deposits for the data storer. The security issue is that the data owner could execute drop attacks by falsely reporting a dispute of a drop attack whilst gaining awards, without having any deposit at stake. For this issue, the design of the system introduces a security mechanism where it allows the data storer to report an attack but does not incentivize the data storer with an award. Lastly, in the timed-release protocol, the reporting mechanism and receiving a reward for correct reporting is separated into two different functions. In our design, we have merged the two mechanisms in one function to save gas costs.

3.4 Off-Chain Behaviour

The smart contract lays the ground rules of our proposed system. It acts as a trusted third party and as a hub for public communication. Off-chain behaviour is code that runs locally on the nodes in a blockchain network. In this section, we will present the most significant algorithms and modules that are used by the off-chain behaviour of the different participants in the system, that is, the data owner, data storer and peers. These are the Whisper protocol module and the algorithms for peer selection, remuneration calculation, and PoS.

3.4.1 Remuneration Calculation Algorithm

The system is designed such that it relies on the service of Ethereum peers. In order to attract peers to contribute to the system, they must be incentivized by the data sender for their service. The total remuneration $\hat{r}$ paid by the data sender to the peers consists of two components which serve their own purpose. The first component $\hat{r_s}$ is paid by the data sender to reward the peers for storing and routing a PoS challenge through the network, while $\hat{r_c}$ is paid to the peers as compensation for invoking smart contract functions as part of their service. The timed-release protocol in [5] employs a remuneration calculation where it lets the $\hat{r_c}$ component be subject to change for different monetary values of the data that is handled by the peers. However, the r_c should only be charged as gas compensation for invocation of smart contract functions, which consequently requires it to be constant. Furthermore, we assume that the monetary value of challenges are constant in contrast to the timed-release protocol, where stored self-emerging data may be valued differently. In addition, as the timed-release protocol allows the formation of only one path, we present a formula for remuneration calculation exclusive for our proposed system.

The $\hat{r_c}$ component is calculated by retrieving the gas costs of the functions in the smart contract, therefore, for k selected peers in a path, $\hat{r_c} = kr_c$, where the r_c is the remuneration corresponding to the gas spent by a single peer. The $\hat{r_s}$ component is considered to be the motivation of the peers to partake in the system. Therefore, it must be sufficient to make the system as an attractive venture. It also has to be higher for longer working windows; that is, peers should be rewarded for every storage hour. Furthermore, since the system benefits from stable and longstanding service, r_s should be larger for a service time closer to the release time t_r than the setup time t_s.

We segment the service times in the calculation in hours. Let $\triangle r_s^i$ be the remuneration of a peer for the i^{th} service hour, and $\triangle r_s^1$ be the first hour remuneration. Furthermore, let α be the per hour increment of $\triangle r_s^i$. That effectively makes $\triangle r_s^i = \triangle r_s^{i-1} + \alpha = \triangle r_s^1 + \alpha(i-1)$. Therefore, we can derive $\hat{r_s} = \sum_{i=1}^{|T^s|}(\triangle r_s^1 + \alpha(i-1)) = |T^s|(\frac{2\triangle r_s^1 + \alpha(|T^s|-1)}{2})$ for the whole storage period $|T^s|$ of a challenge. In total, the expenses of the data storer can be summed up by $\hat{r} = kr_c + |T^s|(\frac{2\triangle r_s^1 + \alpha(|T^s|-1)}{2})$, and the remuneration gained by any peer that serves in the working window $[t_b, t_e]$ in $|T^s|$ is given by $r = r_c + t_e(\frac{2\triangle r_s^1 + \alpha(t_e-1)}{2}) - t_b(\frac{2\triangle r_s^1 + \alpha(t_b-1)}{2})$.

Since we assume that the monetary values of PoS challenges do not differ, the system sets constant values for $\triangle r_s^1$ and α. Therefore, these parameters are set in the smart contract, which consequently means that data owners cannot decide their values. This design choice is made to provide fair remunerations to peers. In Section 5.2, we provide a discussion on what the values of these parameters should be in order to attract peers to partake in the system.

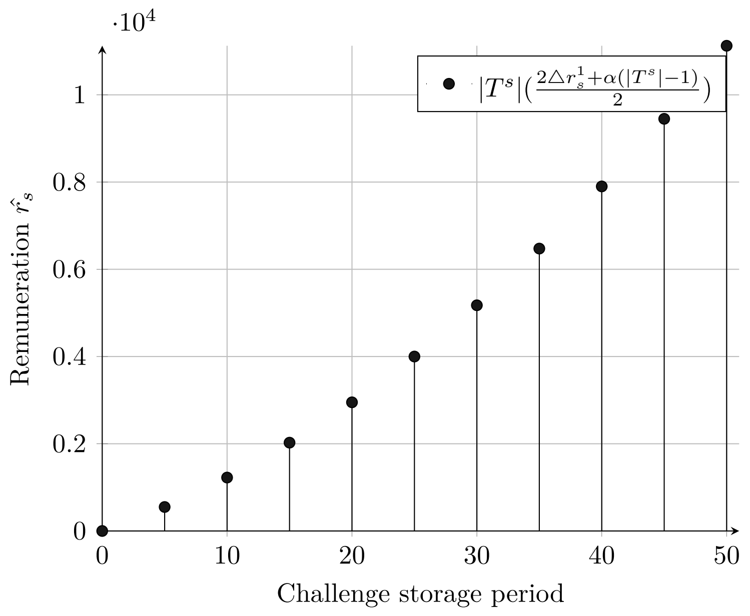

Figure 3.5: Remuneration calculation for $\triangle r_s^1 = 100$ and $\alpha = 5$.

Figure 3.5 shows a plot of the remuneration calculation of $\hat{r_s}$ where $\triangle r_s^1 = 100$ and $\alpha = 5$. It is shown that for the 0^{th} to 10^{th} storage hour the remuneration is 1225, while for the 40^{th} to 50^{th} storage hour the remuneration is 3225. Hence, it is shown that peers are encouraged to serve for long-term storage of challenges. Further analysis and evaluation of remuneration will be given in Section 5.2.

The formulas for calculations of remuneration are similar to the ones presented in [5]. However, we have done the following modifications to make them relevant for our system. Since the compensation for invoking smart contract functions that cost gas should be constant, our formulas, in contrast to the ones presented for the timed-release protocol, do not make r_c subject to change for different values of the routed data in a path. Furthermore, since we assume that PoS challenges do not vary in value, the formulas do not input any variable for the different monetary value of the routed data in a path. In our formulas, the remuneration paid out to the peers is only dependent on $\triangle r_s^1, \alpha$ and $|T^s|$, while r_c is constant. Lastly, since our proposed system allows multiple paths in order to send multiple challenges with different timelines between the data owner and data storer, the remuneration calculation is done separately for each path using the derived formulas.

In addition to our modifications to the remuneration formulas presented in the timed-release protocol, it is of significance to discuss a mathematical error that we have discovered in [5]. Through experimentations with the remuneration formulas of the timed-release protocol, we find that the relationship between the formula for the total remuneration expenses of the data owner and the formula for remuneration gained by peers is wrong. For ease of description of this error, we name the former formula F_1 and the latter formula F_2. If there is one peer in path, the working window is $[t_b, t_e] = |T^s|$. If there are more than one peer, $|T^s|$ is segmented in as many different working windows as there are peers, and each peer receives a remuneration $x = F_2$. The remuneration x is based on the size of peers' working windows and which part of T^s that they cover. Let i represent a peer and let n be the number of peers in a path. The sum of remunerations to all peers in a path, $\sum_{i=1}^{n} x_i$, should equal the result of F_1, but using the formula F_2 derived from F_1 in [5] always gives $\sum_{i=1}^{n} x_i < F_1$. To solve this error, our F_2 that is presented in this section, is constructed such that, for any amount of peers, the sum of the different F_2s is equal to the result of F_1.

3.4.2 Peer Selection Algorithm

The peer selection algorithm used by the data owner effectively selects the most suited peers for a service from the common pool of available peers. In our proposed system, this algorithm is based on the one presented in [5]. However, since we have a system where multiple paths should be allowed for a pair of data owner and storer, the algorithm is modified to account for this. In this section, we give a summary of the algorithm as it is designed for the timed-release protocol, and further present the modifications that are needed to incorporate it to our system.

In [5], it is claimed that the peer selection algorithm should have two objectives, namely minimizing remunerations paid by data owners, and maximizing the expected profit made by peers. Since the remuneration r_s is a fixed amount as discussed in Section 3.4.1, minimizing remuneration paid by data owners could be solved by reducing the number of peers that participates in a path. Furthermore, maximizing the expected profit made by peers can be solved by having a mechanism for reducing the amount of time that their deposit is frozen by unfreezing their deposits as soon as possible. Hence, allowing the peers to reinvest their deposits in new services rapidly. Since the deposit of a peer must be frozen from the start of the service until the end of its working window, it is possible to reduce the amount of time that its deposit is frozen by selecting a working hour closer to the setup time in the case where only parts of its working window is needed for the service. This effectively means that if a peer has a working window of three hours and its service is only needed for one hour; the algorithm should aim to select the first hour rather than the later hours. In [5], the problem that should be solved by this algorithm is segmented into multiple sub-problems, where each sub-problem is defined as follows.

"Given all peer working windows covering an input time point, output the working window that makes the total number of selected peers minimum."

Upon choosing a peer with the right working hour, its end time is used as the input time point for the next sub-problem. This is applied until there are chosen enough working hours from peers to cover the required service time.

Since our proposed system employs multiple paths in contrast to the timed-release protocol, the peer selection algorithm must be modified to account for this. Algorithm 3.5 shows the peer selection algorithm for a single path (line 12-29) similar to the one presented in [5] with the additional modifications that are done in order to incorporate it to our system (line 3-11). The algorithm takes in two main parameters, a set of peer working windows $\widehat{T^w}$, and another set of challenge storage windows $\widehat{T^s}$ that consists of several $T^s = [t_s, t_r]$, representing the paths. It outputs a set of working windows of the selected peers $\widehat{\widetilde{T^w}}$ for the inputted paths $\widehat{T^s}$. The function `selectPeers` loops the set of peer working windows $\widehat{T^w}$ to find the working window T^w that covers the largest interval of T^s possible. That is, it continuously searches for a T^w that has the earliest start time t_b while still covering the current time point that is covered by another working window t_{cur}. This process is repeated until there is a T^w covering the interval T^s. The algorithm traverses $\widehat{T^s}$ and runs the `selectPeers` function for every instance. Then, it subtracts the selected working windows $\widetilde{T^w}$ from the common pool of working windows $\widehat{T^w}$ to prevent that selected peers get selected for multiple paths, and adds $\widetilde{T^w}$ to the set of working windows of the selected peers $\widehat{\widetilde{T^w}}$. The result of this algorithm, $\widehat{\widetilde{T^w}}$, contains the optimum combination of working windows of peers for the requested service periods. The complexity of this algorithm is $O(|\widehat{T^s}||\widehat{T^w}||\widetilde{T^w}|)$.

In addition to the extension of the peer selection algorithm to support multiple paths, we have done a refinement to the `selectPeers` algorithm presented in the timed-release protocol for a single path. Originally, this algorithm contained an additional condition for controlling that a peer was not previously selected before it was added to the set of selected peers. We find that this condition is redundant and unnecessary. If a working window spanning a time period $[t_b, t_e]$ is selected for a subproblem without t_b covering t_r, the next subproblem requires another peer's t_e to be before the t_b of the selected peer.

We have in our algorithm removed this condition since we empirically prove that this is an unnecessary condition.

Algorithm 3.5 Peer Selection Algorithm.

```
Input: Set of registered peer working windows $\widehat{T^w}$, requested challenge storage windows $\widehat{T^s}$ consisting
of several $T^s = [t_s, t_r]$ (representing paths), transfer time period $|T_t|$.
Output: List of working windows of selected peers for all inputted paths $\widehat{\widehat{T^w}}$.

$\widehat{\widehat{T^w}} \leftarrow \emptyset$;
for all $T^s \in \widehat{T^s}$ do
    $\widetilde{T^w} \leftarrow selectPeers(\widehat{T^w}, T^s, |T_t|)$;
    if $\widetilde{T^w} = false$ then
        throw;
    end if
    $\widehat{T^w} = \widehat{T^w} \setminus \widetilde{T^w}$;
    $\widehat{\widehat{T^w}} = \widehat{\widehat{T^w}} \cup \widetilde{T^w}$;
end for

function SELECTPEERS($\widehat{T^w}, T^s, |T_t|$)
    $t_{cur} \leftarrow t_r$; $t_{pre} \leftarrow t_r$; $T^w_{sel} \leftarrow \perp$;
    while $t_{pre} > t_s$ do
        for all $T^w \in \widehat{T^w}$ do
            if $T^w.t_b < t_{cur} + |T_t| \wedge T^w.t_e > t_{cur} + |T_t| \wedge T^w.t_b < t_{pre} + |T_t|$ then
                $T^w_{sel} \leftarrow T^w$;
                $t_{pre} \leftarrow T^w.t_b$;
            end if
        end for
        if $T^w_{sel} \neq \perp$ then
            $\widetilde{T^w} \leftarrow \widetilde{T^w} \cup T^w_{sel}$;
            $t_{cur} \leftarrow t_{pre}$;
            $T^w_{sel} = \perp$;
        else
            return $false$;
        end if
    end while
end function
```

3.4.3 Proof-of-Storage Algorithm

PoS describes protocols that allow a party to verify the integrity of remotely stored data. In principle, a data owner issues a request for proof of storage, commonly known as a challenge, and the data storer proves data integrity by responding correctly to the challenge. Since we have designed a proof of concept system for PoS using self emerging challenges, the implementation of an advanced PoS algorithm is out of the scope of this system design. Therefore, a simple PoS scheme using probing has been designed and utilized. That is, the data owner stores different segments of their remotely stored data and requests the data storer to retrieve the segments. Figure 3.6 shows the simple probing algorithm of our proposed system. The data is segmented in i bytes. Therefore it is possible to retrieve the i^{th} byte of the data using a simple algorithm. A data owner utilizes this scheme to pick random is, representing challenges and runs the algorithm for every i to obtain the answers, as. The different challenges are then issued to the data storer, which again uses the algorithm to obtain the a for every i. It proves data integrity by correctly submitting the corresponding as and is to the data owner.

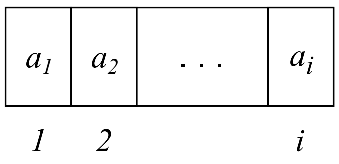

Figure 3.6: Proof-of-Storage using probing.

3.4.4 Whisper Protocol Module

The approach of direct transfer challenges between the participants using the Whisper protocol is chosen since it limits the data transfer between the participants and smart contract, thereby reducing gas costs as discussed in Section 3.5. We have designed an off-chain module that inputs the cryptographic key for the communication session and continuously listens to the network on incoming messages. This module is made such that it is compatible with any of the off-chain modules.

Algorithm 3.6 shows the behaviour of this module. It offers a set of functions that use the API of the Whisper protocol to enable direct transfer of data between participants. The function `newWhisperSender` generates a symmetric Whisper key which the invoker can use to encrypt using the public key of the intended receiver and submit to the smart contract for the receiver to download. Function `sendMessage` inputs a symmetric Whisper key *symKey* and the data to be transferred *msg*. It allows the invoker to send data to the intended receiver that possesses the same symmetric Whisper key. A topic for the communication is set by `w.setTopic` since the API requires that a topic for the communication set. We assume that for all communication using the Whisper protocol in this system, there is a consensus between all participants about the value of the topic. Furthermore, the function creates a client using `w.connect`, creates an encrypted message using *msg*, *symKey* and *topic* as input parameters to `w.newMessage`, and lastly transfers the message into the network using the created client. A receiver can create a channel for incoming messages using the `newWhisperReceiver`. This function requires the symmetric Whisper key *symKey* for the communication session. It sets the topic of the communication the same as the sender and creates a channel for incoming data using `w.subscribe`.

Algorithm 3.6 Whisper Module

```
Uses: Whisper API, $w$;
$sender \leftarrow \perp$;
$posTopic \leftarrow$ "Proof of Storage";

function newWhisperSender
    $symKey \leftarrow$ generateSymmetricKey()
    return $symKey$
end function

function SendMessage(msg, symKey)
    $topic \leftarrow w.setTopic(posTopic)$;
    $client \leftarrow w.connect()$;
    $message \leftarrow w.newMessage(msg, symKey, topic)$;
    $client.Post(message)$;
end function

function newWhisperReceiver(symKey)
    $topic \leftarrow w.setTopic(posTopic)$;
    $listener \leftarrow w.subscribe(symKey, topic)$;
    return $listener$;
end function
```

In the timed-release protocol, it is stated that Whisper protocol is used for P2P communication between the participants in a path. However, there is no description provided in what way this off-chain behaviour is conducted. Therefore, the module for direct communication between the participants is created exclusively for the system proposed in this thesis using the Whisper API.

3.5 Alternative Designs

This section selects some elements of the system's design and presents and discusses alternative approaches. The alternative designs are not necessarily improvements in all cases, but rather interesting approaches that could be used if other assumptions or new requirements are set to the system.

The current design of the system allows peers to be selected only in a single path. Moreover, it does not allow peers to be selected to serve in paths of multiple data owners.

This could have been implemented into the design by allowing a peer to register multiple deposit accounts. This could also have been solved by only freezing parts of the deposit as requested by a data owner. Although this is a fairly simple design concern, for the sake of simplicity, it is not included in the design of the proposed system.

Another alternative design could be to explicitly protect the system against drop attacks from data storers. The current design assumes that data storers are externally motivated to honestly follow the guidelines of the system, and correctly answer the received challenges. Moreover, the design does not incorporate payment of deposit from data storers since the goal is to allow data storers to partake in the system without the need to adapt their already implemented and well-established infrastructures for storage and payments. If deposits were required, data storers would have to account for additional expenses related to participating in this PoS system. In an alternative system where this does not apply, or explicit protection is needed, the system could require a deposit from the data storer that is the same deposit that is paid by peers multiplied with the number of paths in the service. This way, the same protection against drop attacks as in the timed-release protocol is offered.

The design of the system is such that the transfer of the encrypted PoS challenges between the participants is done through the P2P communication protocol, Whisper. An alternative approach of transferring the challenges would be to let each participant submit the challenge to the smart contract. However, such an approach would result in participants later in the path having access to the challenges. Although the challenges are encrypted with public keys of the peers and data storer, in an extreme case where one or more private keys are compromised, the later participants could potentially decrypt a challenge and execute a release-ahead attack. Another drawback of this alternative approach is that the amount of data that would be transferred in a process would be considerably larger than the current design. For each time a participant decrypts a challenge to obtain the certificate, it would have to submit the challenge concatenated with the remaining certificates of the future participants in the path. For a challenge c concatenated with n certificates d, this would result in n submissions of the challenge, and $\sum_{i=0}^{n}(n-i)$ submissions of certificates d where i represents the i^{th} participant in the path. This results in a total of $\sum_{i=0}^{n} c + d(n-i)$. However, using the Whisper protocol restricts the submission of data transfer to only be $n \times d$ since the challenge c does not get submitted to the smart contract, and each peer only submits its own certificate d.

Throughout the description and discussion of the design, it is assumed that the system is implemented on top of the Ethereum platform. However, any blockchain platform that supports smart contracts and transactions could be used by the proposed system. It would be interesting to deploy a blockchain only for the purpose of decentralized storage. For this, our proposed system could be employed, or even integrated into the platform, for self-emerging PoS challenges. In addition, it is conceivable that the payment to data storers in such a decentralized storage system would be incorporated into the system. Therefore, it would also be possible to enforce explicit protection of drop attacks from data owners by requiring them to pay a deposit, as previously discussed in this section. In fact, this would be a more natural approach in this case compared to the current design since the payment for the subscription could indirectly be used as data storers' deposit. That is, the data storers would not get paid for their services from data owners if they cannot prove the integrity of the owners' data by honestly following the ground rules of the system and correctly responding to PoS challenges. Lastly, if a P2P communication service equivalent to the Whisper protocol were not offered by the blockchain platform, as previously discussed in this section, the design could be altered so that each participant would submit the challenge to the smart contract.

Chapter 4

Test Setup

This chapter presents a test setup that was created for the purpose of experimenting with the relevant functionality that our design employs. The off-chain behaviour of the participants was not fully developed due to the time constraint of this thesis. Therefore, the test setup was not used for experimental evaluation. However, the test setup facilitates this. The purpose of this chapter is to support our work by providing credibility that the different functionalities have been properly tested. Both the module for P2P communication through the Whisper protocol and various, relevant off-chain behaviours have been tested in an instance of the Ethereum blockchain platform. Furthermore, the implemented smart contract that employs our design has been deployed in this network, and we have tested each function in different service configurations to obtain gas costs. In this chapter, we present the various test environments that were assessed. Additionally, we describe how a private, local Ethereum blockchain was created using Docker for comprehensive testing and experimentation of the implemented smart contract.

The Ether needed for execution of transactions on the main Ethereum network needs to be bought with real currency. Therefore, several alternative systems for deploying and testing smart contracts exist for experimentation. These include both local and public solutions. An example of a promising local solution is Ganache [14] which allows for fast setup of a personal Ethereum blockchain that can be used to run tests, execute commands and easily inspect the state of the blockchain. There also exists some official Ethereum public test networks. Although the Ether is already mined in many of the test networks and can be gained for free using Faucets, the level of control of the blockchain is not comparable with a private, local blockchain as offered by Ganache.

For experiments of our proposed system, both local and public test networks were considered. A popular public test network, Rinkeby [15], was attempted to be used in the early stages of this project. However, the chain data of this blockchain is large, and therefore, it was excluded as an option since it requires a considerable amount of storage space. The choice of setup was therefore restricted to a local Ethereum network. For this, Ganache was used for development and testing of this system's smart contract.

In this project, the secret information, the challenges, are sent through the Whisper protocol which is a P2P communication protocol for Ethereum nodes. Since the proposed solution makes extensive use of the Whisper protocol, it was essential to create a network of separate nodes that ran and participated in an Ethereum network. In the process of implementing the functionality of the system, it was discovered that Ganache did not support the Whisper protocol [16]. Therefore, an alternative setup for experimentation was necessary. For this, we used Geth [17] on separate nodes to form a blockchain network. Geth is the official Golang implementation of the Ethereum protocol. In the following section, we present how this was achieved.

4.1 Setup of Local, Private Ethereum Network Using Docker

The final setup for experiments was done on a local, private Ethereum network using Geth and Docker [18]. Although many of the smart contract functions could be tested as their own unit, it was necessary that the Whisper protocol was included in the testing so that the system could be tested as a whole. The most realistic setup would have been to deploy the system on a public test network, but because of the aforementioned drawback of this setup and the limited resources available, this setup posed a challenge. Another realistic setup would have been to deploy a testbed of separate nodes, each on a dedicated machine, running Geth to form an Ethereum network. However, this setup would also have posed the challenge of high costs due to significant resource demands. Since Docker allows for several containers running on the same machine, it was possible to create an Ethereum testbed with several nodes running on a single machine. Although there exist some Docker setups for running a private Ethereum network, at the time of developing this system, there were none found that were satisfactory and additionally ran the Whisper protocol.

Prior to entering the details of the setup for the experiments, it is necessary to comprehend some of the features of Docker that has been extensively used for this project. Therefore, we will present an overview of the features that are relevant for the setup of the private Ethereum network used for experimentation of the system. To create containers in Docker, a Docker image must be specified. Images are modular, which means that images can build upon each other. In one image, it is possible to specify that it should inherit another image, called a base image. This functionality allows to place common configurations that are needed for different images in one base image. Thereby increasing efficiency and avoiding mistakes that are difficult to detect. Figure 4.1 shows the structure of a Docker setup where an image inherits a base image of common configurations which again inherits another base image containing the operating system, Arch Linux. After the configuration of an image, it is possible to use it to create a virtual machine, called a container. The container will then possess configurations and installations that are specified in the image.

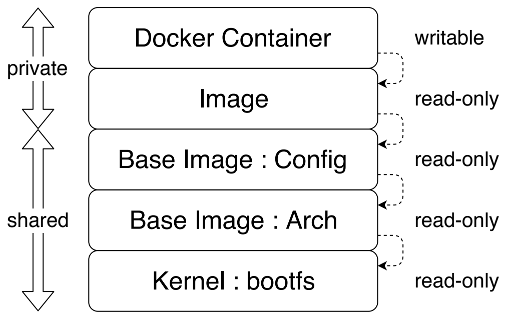

Figure 4.1: The connection of Docker images and containers.

For the experimental setup, a base image was created to specify all of the necessary, common configurations for the nodes in the network. This included specifying the operating system, installing the needed tools, and adding a user to run the Ethereum node. Furthermore, the private Ethereum network was initialized using Geth. The main attributes of the network were configured by creating a genesis block. While new blocks in the blockchain are continuously generated in an established blockchain network, the first block, called the genesis block, needs to be specified. Listing 4.1 shows the genesis block with its attributes that were created for the Ethereum network used for the experiments in this project. The most noteworthy attributes in the context of the test setup are the difficulty and allocation attributes. The difficulty is set low to save computing resources for the miner node. The allocation attribute is used to set an initial balance for some of the nodes partaking in the Ethereum network.

```
"config": {
    "chainId": 13,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
},
"coinbase": "0x0",
"difficulty": "0x2",
"nonce": "0x0000000000000097",
"mixhash": "0x0",
"parentHash": "0x0",
"extraData": "0x",
"gasLimit": "0x8000000",
"timestamp": "0x0",
"alloc": {
    "ec6a4e03bedaaac4c6070e8a96f19de71fb874e2": {"balance": "30000000000000000"},
    "8090726167de701826cf63fdb560bda7ae652be4": {"balance": "30000000000000000"},
    "2828a8f260baa874625b3b794ed6509225f48b40": {"balance": "30000000000000000"},
    "ef8383c0ef7a45eb1b956b087e7dc59d0706b274": {"balance": "30000000000000000"},
    "4bccda700fd05bfcf553fdaafcaaaee5e772a27e": {"balance": "30000000000000000"},
    "f71195df0eb8438dd9eaf839d3abb47859f2175f": {"balance": "30000000000000000"}
}
```

Listing 4.1: Genesis block configuration

Subsequent of creating the base image, specific images for different types of nodes with their respective configurations were created. These included bootstrap, miner, explorer and several regular nodes. The bootstrap node is a node that is selected to stay connected to the Ethereum network. Its role is to act as a gateway for new nodes to connect and partake in the network. The role of the miner node is to allocate its resources for mining, that is processing transactions, in the Ethereum network. The explorer node opens an HTTP port in order to host a web site that shows a continuously updated overview of the Ethereum network including recent blocks and transactions [19]. The regular nodes are nodes that only partake in the Ethereum network. These were used to run and test the off-chain code of data owner, data storer and peer.

To configure the nodes collectively, Docker Compose was utilized. This is a tool that enables the creation and execution of multiple Docker containers. There were several properties of the network that was specified in the configuration of Docker Compose. These included IP addresses, ports, and input data for each container. To enable communication between the nodes, the definition of a network and allocation of IP addresses need to be specified in the configuration. The ports that needed to be open for communication and the mapping between inbound and outbound ports also needed to be specified. Furthermore, there were a set of input data for each container that was required to make the network function properly. These mainly included the bootnode IP and ID, node key and key-store files.

In the development process, it is useful that the Ethereum address of the nodes are deterministic. Since the address of a node is used in every aspect of interaction with the Ethereum network, such as communication with smart contracts and transferring Ether, the development process would be cumbersome if the address was not static for every time a node is booted up. Therefore, a set of accounts were created using the Geth tool and set into the configuration of each node. The accounts are given as key-store files. Listing 4.2 shows an example of a key-store file of an account. Here the most noteworthy attributes that are generated are the Ethereum address and the cipher related attributes. Furthermore, the private key of that address will be static when the cipher related attributes of an address are statically defined. This is useful since the private key is required for authentication in any transaction with a smart contract that involves the transaction of data. The web3.js library, which allows interaction with Ethereum nodes, can be utilized to obtain the private key from the key-store file together with the password that was used upon creating the file.

```
"address":"8090726167de701826cf63fdb560bda7ae652be4",
"crypto":{
  "cipher":"aes-128-ctr",
  "ciphertext":"03936c7cd882e97fe72431af1b89077601d1a8de242534adad05133eb931fb69",
  "cipherparams":{ "iv":"2c29f7446b18715961c1c1022ae163c0" },
  "kdf":"scrypt",
  "kdfparams":{
      "dklen":32,
      "n":262144,
      "p":1,
      "r":8,
      "salt":"c3b1c08e49564acaf9576dcafb00f4bf0a3a7a48190a1df8483e6284cdbb3ebf"
      },
  "mac":"5b9549f0e43b97e4aa6cf7dde163b20ca472b9f1b7651148f75c2be81034b0df"},
"id":"8494c847-247a-4b25-8fee-9f14bfd38d88",
"version":3
```

Listing 4.2: Key-store file of an Ethereum account.

Establishment of P2P connection between nodes does not happen automatically. In the Ethereum network, each node has an ID, called a node key. Connecting to other nodes can be done manually by adding their IDs through the Geth console. The ID of the node is created and stored in a file by connecting to the network, and therefore, it is deterministic as long as the file does not change when connecting to the network again. However, spinning down the Docker cluster of nodes erases all data, which poses a problem. To solve this, the node keys was given as an input to Geth when connecting to the network, which always makes the ID of the node deterministic. Furthermore, interconnecting the nodes can be done by adding a file called "static-nodes.json" in the Geth data folder. By default, Geth derives the IDs of the nodes that are specified in the file and establishes a P2P connection with them when connecting to the Ethereum network.

4.2 Implementations

A smart contract that employs the proposed design was fully developed and tested in multiple platforms, including the local Ethereum network. However, due to the time constraint of this project, the off-chain code, that is the local behaviour of the different participants in the system, was not fully developed. However, relevant parts of this off-chain code were developed using the Go programming language and tested in a local Ethereum network. This included peer registration, data owner sign up and the Whisper protocol module. The Whisper protocol module, together with the configuration of the local Ethereum network, can be found in Appendix A.

4.3 Testing Smart Contract Code

In the development of this system's smart contract, testing was essential. For this, the Truffle suite [20], which offers an automated testing framework, was used. While Truffle offers the possibility of writing unit tests in both Javascript/Typescript and Solidity, the system's unit tests were written in the latter language. This is because the implemented smart contract that employs the design has been written in this language. Continuous unit testing was the preferred methodology as it increases efficiency while developing and debugging the smart contract since the set of tests can easily be executed upon changes in the smart contract. The unit tests also eliminate the need of deploying the contract to the network and executing client code which makes it very useful in the early stages of the project where the contract is written prior to implementing any client code.

Chapter 5

Evaluation and Analysis

In this chapter, we provide evaluation and analysis of the implemented smart contract that employs our design for self-emerging PoS challenges. First, we evaluate the gas costs of the implemented smart contract based on experimental results. Then, we provide a discussion and evaluation of the remuneration paid to partaking peers in the system where we compare their gain here with other alternatives. Further, we analyze the different security aspects of the system. Lastly, we present a set of possible future directions.

As discussed in Section 2.2.2, the performance of a smart contract is governed by the performance of the underlying blockchain platform. In the Ethereum platform, the block time varies between 10-20 seconds [11]. Therefore, measurements such as throughput and latency to evaluate the performance of the system yield results that mostly portray the performance of the blockchain platform. As a consequence, we do not evaluate the proposed system based on these measurements, but rather base our evaluation on gas costs. The gas costs are obtained from our experimental results, where we have implemented and tested a smart contract that employs our proposed design. This smart contract will be referred to as the self-emerging challenges (SEC) contract in the following sections.

5.1 Gas Cost

In Ethereum, gas is required to pay for the execution of smart contract code inside the Ethereum Virtual Machine. Therefore, it is essential that the gas cost is considered when designing a smart contract. The experimental results in Appendix B show the gas costs of the main functions of the SEC contract for different configuration setups. The gas costs that are presented were obtained by setting up various scenarios with different amounts of PoS challenges and different amounts of peers in paths.

Gas costs are subject to change. Several factors can affect the outcome of the costs. In the Ethereum yellow paper [10] it is described that gas usage increase by the number of operations and the amount of data that is sent to the smart contract. Therefore, the functions in the SEC contract with high gas usage are either subject to high storage utilization, many operations or both. As an example, the update functions are all functions that take input parameters and update an attribute of the function caller. Since the input data and consequently, the size of data varies, the gas usage is different from each update function. Furthermore, the gas usage of a single function may also vary. In cases where the input parameter of a function is a datatype where the size is inconstant, e.g. string, bytes or arrays, the gas cost can drastically vary. For example, the gas usage of function `newPeer` for an RSA public key with a key size of 512 has been tested to be 178 325 units of gas. However, for a public key with key size 2048, the function requires 325 062 units of gas. These numbers are derived from experimental results (see Appendix B).

The gas cost of executing a smart contract code is a product of gas units and gas price. Upon construction of a transaction, the caller sets the gas price of the transaction, that is, how much Ether the caller is willing to pay per gas unit. The gas price is what determines how fast a transaction will be mined. This is because a miner is more eager to mine a transaction that is better paid. As of date, 2019-04-29, the average gas price in the Ethereum main network is 1.035×10^{-8} Ether [21] and the Ether price is $157.49 [22].

5.1.1 Gas Costs Analysis of the Smart Contract's Functions

Figure 5.1 shows a plot of the monetary cost of each function for a configuration with one peer in one path. As is evident from the plot, most functions cost under $0.2. The exceptions are `newPeer`, `storerSign`, `setup` and `setProps`. We will in the following analyze the gas costs of the functions.

It is tested that the `newPeer` and `storerSign` functions do not change drastically for different setups. This is because the costs of these functions are mostly dependent on the data transfer, not varying number of operations. Although the same applies to the

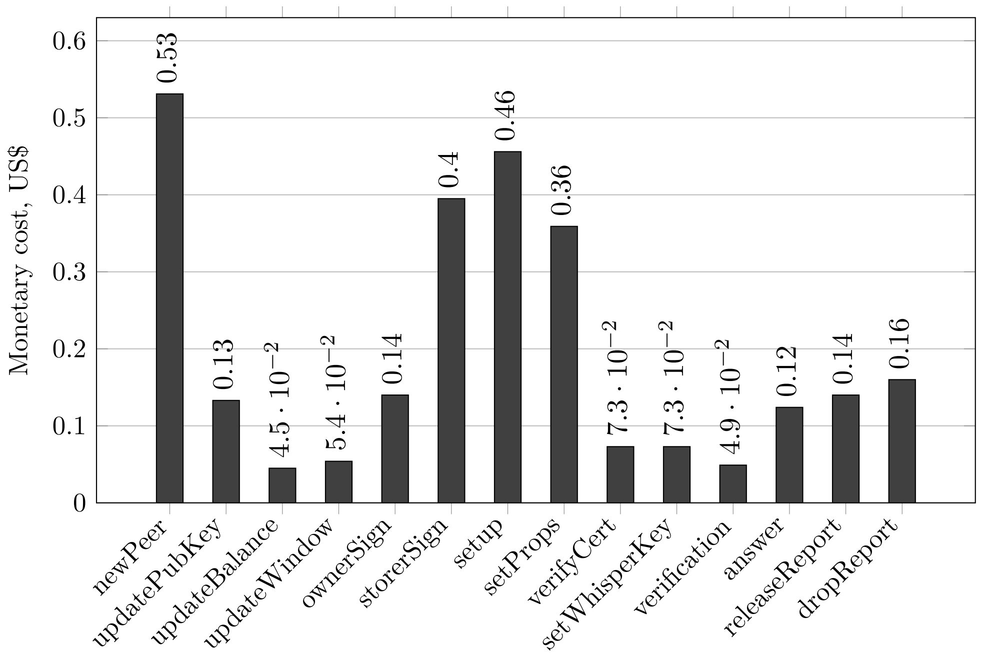

Figure 5.1: Monetary cost of each function. Calculated from gas measurements of a setup with one peer in one path and 2048 bits public keys (Table B.1). Calculations are based on gas price 1.035×10^{-8} [21] and Ether value US$157.49 [22]. For functions `verifyCert` and `dropReport` that can be invoked by both peers and storers, the highest gas measurements between the two are chosen.

cost of the `setProps` function, it is important to note that the cost of this function is dependent on the number of selected peers and paths. That is, for larger amounts of paths and peers, the system requires the data owner to submit more data through this function.

The `setup` function is subject to many operations. For example, this function traverses the selected peers to check if they fulfill the requirements and freezing each peer's deposit. It also checks if the data owner has transferred sufficient amount of remuneration by executing the remuneration calculation for each peer. In the SEC contract, this function requires the addresses of the selected peers, information about the formations of paths, and information about payment. The largest amount of data is the addresses of peers. However, each address is small compared to a public key that is submitted, for example, in `newPeer`. Therefore, most of the cost related to this function is due to computations. Moreover, since the number of computations is dependent on the input parameters, the costs increase as the size of the input parameters increase. This can be seen for different configurations in Appendix B.

The sum of the monetary costs of the peer functions is \$0.96. This includes all four functions in the peer registration protocol and the three functions in the service enforcement protocol that are invokable by peers. The update functions in the peer registration protocol are not mandatory to invoke, and therefore, we exclude their cost from the evaluation. This reduces the overall gas cost the peer to \$0.73.

The data owner functions are only called once, and they sum up to \$0.96 for this setup. However, it is important to note that its functions are dependent on the number of paths and peers. An increase in the number of paths and peers results in high data transfer, which then results in higher gas costs.

The sum of the monetary cost of the data storer functions is \$0.59 for this setup. However, if there are multiple paths in the system, i.e. more than one challenge, then both `verifyCert` and `answer` need to be invoked in each path. Therefore, this sum will increase for more challenges. The `verifyCert` function seem to be approximately constant and independent of the number of paths and peers for a single invocation (see Appendix B). In contrast, the function `answer` traverses every peer in a path to pay out remuneration and unlocking deposits. Its cost should, therefore, increase with the number of peers in a path. However, from the test results, this does not seem to affect its cost significantly.

5.1.2 Gas Costs Analysis of Configurations with Multiple Paths and Peers

In the following, we use the plot Figure 5.2 as an example to another service with three paths where each path contains three peers. This example is used to discuss the overall gas expenses of the data owner, which includes compensation to the gas used by the other participants. We assume that there will not be any misbehaviour. That is, the functions in the reporting protocol are not invoked and verifications are successful. For this setup, the total gas expenses of functions invoked by the data owner are \$2.49. The cost related to compensate all nine peers for invoking gas functions is \$6.57. If we assume that the expenses of the data storer need to be paid by the data owner, either directly or indirectly through an increase in the storage service fee, the expenses of the data storer need to be added to the total expenses of the data owner. For this setup, the monetary value of all invocations of the required function by the data storer equals to \$0.8. In summary, the total monetary expenses of the data owner for setting up a service with the described configuration yields $\$2.49 + \$6.57 + \$0.8 = \$9.86$.

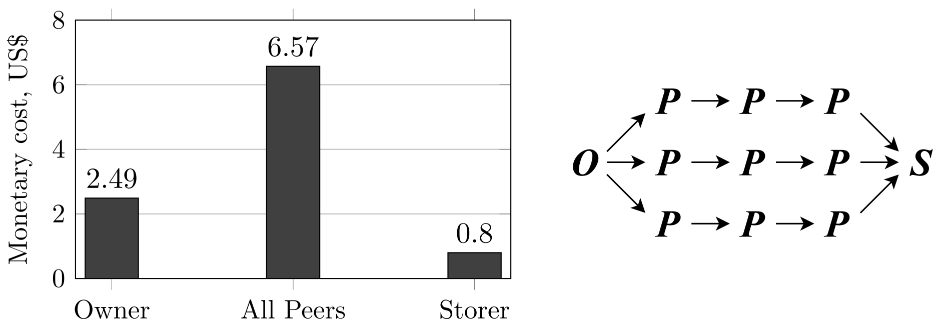

Figure 5.2: Monetary gas expenses of different participants for a setup with three paths where each path contains three peers. Calculated from gas measurements in Table B.3 and B.6. The tables are combined since the number of paths does not affect the gas costs of the missing functions in Table B.6. Calculations are based on gas price 1.035×10^{-8} [21] and Ether value US$157.49 [22]. The reporting and update functions are not included.

It is evident that the greatest cost is due to the number of peers in each path. Figure 5.3 shows the monetary gas cost related to the functions that the participants have to invoke in a setup with three paths where each path contains one peer. We use this setup to show that reducing the number of peers while still issuing the same amount of challenges as in the previous example significantly reduces the total monetary expense. This setup yields $2.15 for the expenses of all peers, $0.8 for the expenses of the data storer and $1.59 for the expenses by the data owner itself. If a data owner should compensate the data storer in addition to the peers, this setup costs the data owner $4.54 in total. This is less than half of what it would cost if all paths contained three peers instead of one. Therefore, we can conclude that it is crucial to minimize the number of selected peers to reduce gas costs.

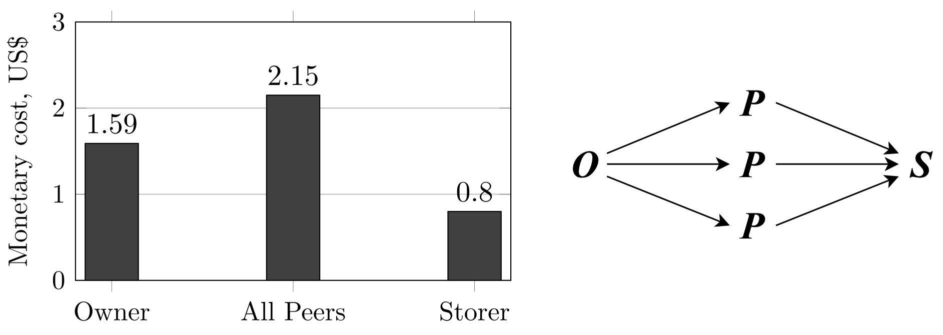

Figure 5.3: Monetary gas expenses of different participants for a setup with three paths where each path contains one peer. Calculated from gas measurements in Table B.5. Calculations are based on gas price 1.035×10^{-8} [21] and Ether value US$157.49 [22]. The reporting and update functions are not included.

Besides striving to include the least amount of peers in a path, a reduction in gas costs can be achieved by limiting the data sent to the contract and reducing the number of operations and calculations by the smart contract. Previously, it was discussed that there is a significant gain in reducing public key sizes from 2048 bits to 1024 bits. Although this reduces the security of the RSA public key encryption, this is a possible method of reducing the amount of data sent to the smart contract. Another method to achieve this is by assigning peers a short ID upon registration and only inputting their IDs upon peer selection rather than their longer Ethereum addresses. The reduction in the number of operations and calculations by the smart contract can be achieved by developing more efficient code.

The SEC contract allows a data owner to issue multiple challenges by allowing to set up multiple paths between the data owner and data storer. In the following, we explore the reduced cost of establishing three paths versus establishing three separate services for delivering three challenges. This is a critical evaluation since we have done extensive additions and modifications to the timed-release protocol, which the design is built on, to allow multiple paths between the data owner and data storer. In Section 5.1.1, it is shown that the monetary gas costs of setting up a service with one path containing one peer are \$2.28 in total for all participants. Therefore, the cost is \$6.84 for three separate services. The plot in Figure 5.3 shows that the monetary gas cost of a setup with three paths where each path contains one peer. For all participants, this cost is given as \$4.58, which is 33% cheaper than setting up three separate services. Hence, we show that our proposed design greatly reduces the total costs in the system by allowing the setup of multiple paths.

In the following, we present a comparison of the gas costs in the SEC contract and the gas costs in the timed-release protocol [5]. Although the design of our proposed system is based on the timed-release protocol, we have done extensive changes to allow self-emerging PoS challenges and submission of proof of data integrity by data storers. Furthermore, since [5] does not contain any detailed description of the implementation of the timed-release protocol, we have built our system from the bottom up. Therefore, even for functions that are similar and effectively have the same tasks, their approaches may be different. Hence, the two systems are not directly comparable. Nevertheless, we provide some insight and comparison of the gas costs of these two systems for two of the functions that should be directly comparable. The most notable difference in function gas cost is the `updatePubKey`. We have in our design chosen the public key size to be 2048 bits, which costs 81307 units of gas for this function. The corresponding function in the timed-release protocol costs 31785. Since the purpose of this function is only to assign a public key to a peer, this difference in gas cost can be explained with a different choice of key size.

5.2 Remuneration

Deposits are needed in order to prevent misbehaviour in our proposed system. It is just as important to reward peers for their service in order to attract Ethereum peers to register to the protocol. In this section, we will discuss the remuneration given to Ethereum peers that partake in our proposed system. Based on the remuneration formulas from Section 3.4.1, we create an example and evaluate it against interest rates in stock investments. Since peers are attracted to the system due to the remuneration received after correctly completing their service, the remuneration should mirror the amount of return that they would get by placing their deposit in stocks. Moreover, since we assume that data owners pay a fee to their respective data storers for their service, the assurance of data integrity that our proposed system offers will be added to their total expenses. Therefore, it is in the best interest of the data owners that the total amount of remuneration of peers is held as small as possible. A middle-ground between the two aforementioned considerations should be found in order to make the system attractive for both data storers and peers.

In Section 3.4.1, we derived formulas for calculation of remuneration. Equation 5.1 shows the derived formula for calculation of the total remuneration expenses of the data owner, while Equation 5.2 shows the reward paid to a single peer for its working window. In Equation 5.2 we have excluded the r_c component which is paid to each participating peer. This is because it only acts as compensation for invoking smart contract functions that require gas. However, this component is a considerable part of the total remuneration expenses of the data owner, and therefore, it will be included in Equation 5.1. From Section 5.1 we know that this component is tested to be approximately \$0.73 for any setup.

$$\hat{r} = kr_c + |T^s|(\frac{2\triangle r_s^1 + \alpha(|T^s| - 1)}{2}) \tag{5.1}$$

$$r_s = t_e(\frac{2\triangle r_s^1 + \alpha(t_e - 1)}{2}) - t_b(\frac{2\triangle r_s^1 + \alpha(t_b - 1)}{2}) \tag{5.2}$$

In [5], the deposit that is needed to be paid by peers in order to prevent misbehaviour in the system is set to reflect the value of the self-emerging data. In our proposed system, the transferred data are PoS challenges. In our proposed system, the data owners should set the required deposit to reflect the data owner's gas expenses of setting up a service. Hence, if a peer misbehaves, its deposit will be transferred back to the data owner as coverage for the failed service. In Section 5.1.1, the gas costs of a data owner were found to be \$2.28 for a setup with one path where the path included one peer. In the following example, we will use a deposit of \$3 for simplicity.

We continue our discussion of the reward to peers with its analogy of placing the deposit in stocks. Furthermore, we use the interest rate of the American stock market index, S&P 500 Index, as a basis for the following example. The average annual interest rate for the S&P 500 Index is roughly 8% [23]. Therefore, for a peer to participate in the system, the remuneration r_s should be competitive to stock returns with the same deposit and this interest rate over the same course of a period. Equation 5.3 shows a formula for the built up interest on a deposit d^s with an interest rate i over the course of T_{months} months. Figure 5.4 plots Equation 5.2 with set values of $\triangle r_s^1$ and α, and Equation 5.3 with deposit $d^s = \$3$. In the figure, it is shown that when the first-hour pay and the per hour increment are chosen to be $\triangle r_s^1 = \$3 \times 10^{-5}$ and $\alpha = 10^{-8}$ respectively, stock returns are smaller than the remuneration paid by the system in the case of 8% interest rate. Nevertheless, this is not the case for any values of d^s and i. For other cases, $\triangle r_s^1$ and α should be chosen so that the r_s yields a competitive remuneration for given values of d^s and i at any point in time. This effectively means that if the data owner chooses the required deposit of peers differently, or the chosen interest rate is not considered competitive, then both $\triangle r_s^1$ and α need to be adjusted.

$$r = d^s(1 + i/12)^{T_{months}} - d^s \tag{5.3}$$

The plot in Figure 5.4 shows that for the chosen values d^s, $\triangle r_s^1$ and α, the remuneration to a peer is \$0.127 for partaking in a service for four months. Moreover, the bonus that is paid to encourage peers to serve for long-term periods, that is α, affects the remuneration greatly in this case. In contrast, if the peer would place the same deposit in stocks, its deposit would have grown by \$0.081 given 8% interest rate.

In the following, we make an example of the total remuneration that needs to be paid to peers for a setup of twelve paths where each path contains one peer. The number of paths is chosen so that the data storer can receive a challenge once every month for a year. Given the same $\triangle r_s^1$ and α as the above example, the total expenses related to remuneration to peers, as given by Equation 5.1, yields $\sum_{T^s=1}^{12} \$0.73 + |T^s|(\frac{2\triangle r_s^1 + \alpha(|T^s|-1)}{2}) = \$12.1$. Let Dropbox be a data storer in the system. Dropbox's Plus storage plan has an annual

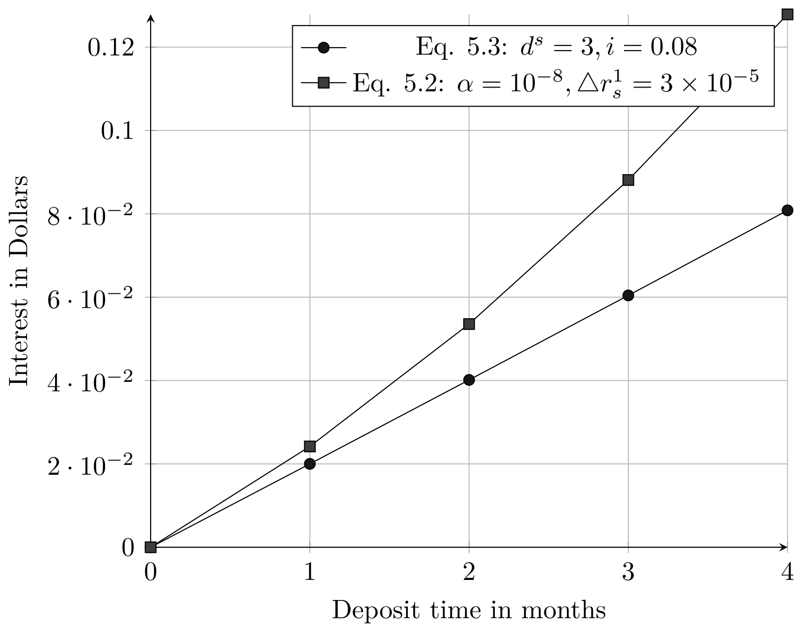

Figure 5.4: Plots of Equations 5.2 and 5.3.

fee of 99\$ [24]. In comparison, the total peer remuneration expenses, including the compensation factor, of the data owner are significant to the annual storage fee at Dropbox. For this setup, given that a data owner uses Dropbox as its storage service provider, the expenses of remuneration to peers gives an increase of 12.2% relative to the annual fee the Dropbox's Plus storage plan. In Section 5.3, we give an example of the total expenses of the data owner including its gas costs, compensation of used gas costs by other participants and remuneration.

The Ether value is subject to change. In a period of two years, April 2017 to April 2019, the minimum Ether value in U.S. dollars was 141\$, while the maximum reached 1100\$ [25]. This level of fluctuation is not atypical for cryptocurrencies [26]. Since the level of fluctuation can be high, the profitability in participating in the network may vary for peers. Furthermore, since peers freeze their deposit from setup time to release time of a challenge, they cannot trade their Ether if its value is decreasing during a service. By the time the smart contract unfreezes a peer's deposit and pays remuneration, this amount of cryptocurrency may be worth severely less than what the peer anticipated. More secure participation by a peer, in terms of risk, is offering its service for short periods of time. To prevent this, the remuneration formulas offer more remuneration for service hours closer to the release time of a challenge, than the setup time. Therefore, a data owner should aim to set up services that have limited release time of challenges. If release time over longer periods is necessary, the $d^s, \triangle r^1$ and α must be set to account for this.

That is, peers should receive a remuneration that makes up for the risk of freezing their deposit over longer periods.

In Section 5.1, we concluded that utilizing exactly one peer in each path significantly reduced the total gas costs in the system. In this section, we support this conclusion by showing that the utilization of only one peer in a path additionally provides a more fair remuneration payout to peers. Let a path consist of two peers where a self-emerging challenge has a release time two months after the setup time. The first peer provides its service the first month, while the second peer serves the last month. Using Equation 5.2, we calculate the remuneration of the first peer to be \$0.0242, and the last peer to be 0.0294. The last peer's remuneration is over 21.5% larger in order to encourage peers for long-term service. However, the design freezes the deposits all peers until a challenge is delivered to the data storer, and it is proved that the peers have not misbehaved. Since both peers in this example have their deposits frozen for the same amount of time, it is unfair that the last peer receives more remuneration than the first peer. Therefore, we conclude that the utilization of exactly one peer per path will optimize the fairness of remuneration payout.

5.3 Summary of the Total Expenses of the Data Owner

In Sections 5.1 and 5.2 the different expenses of data owners were evaluated. The total expenses of data owners are the sum of remuneration and gas compensation to selected peers and the gas used by data owners themselves. If we assume that the data storers adjust their service fee to include the gas used in this system, the expenses of the data owners have to include the gas used by data storers. We will, in the following, give an example of the total expenses of a data owner with a realistic scenario using a setup of three paths where each path contains one peer. In this scenario, each challenge is delivered ten days apart. This consequently means that the first peer holding the first challenge needs to serve for ten days, the second for twenty days and the third for thirty days. In the data owner's total expenses, we include gas usage by the data storer and the data owner itself, and gas compensation and remuneration to peers. In Section 5.1 for the same setup, the gas compensation for all peers were found to be \$2.15, while we calculate their remuneration to be $\$0.0075 + \$0.0155 + \$0.0242 = \$0.0472$ using the same d^s, $\triangle r_s^1$ and α as in Section 5.2. The gas used by the data owners and data storers were \$1.59 and \$0.8 respectively for the same setup. Therefore, the total expenses of the data owner for this setup are \$4.59. Compared to Dropbox's monthly fee, which is \$8.25 [24], this yields 55.6% in additional expenses for data owners given that they issue three PoS challenges a month. Of course, each data owner has the possibility to issue more or fewer challenges, thereby affecting the cost related to utilizing this system.

5.4 Security

This section details a security evaluation of the system. We will discuss the different types of attack that can be performed and show that the design of the system makes it disadvantageous for rational participants to perform these attacks. Furthermore, we evaluate the affected security of only utilizing and allowing one peer per path.

False Challenges

The design assumes that the challenges and the corresponding answers submitted by the data owner are correct. Meaning that the data owner has not submitted fake challenges for the stored data. This assumption is made to focus on the objective of this thesis; making a system for self-emerging PoS challenges. Therefore, the design assumes that public verification of data integrity is available. As presented in Chapter 2, public verification is a property of a PoS scheme where it allows a third party auditor to verify the integrity of data. A solution for this could be done by having the data owner and data storer to agree on the hash of the data before the PoS process. Then, if there is a dispute between the two parties, the data storer could escalate the situation to publishing the data to a third-party auditor. This is certainly a costly operation, but it serves its purpose. There exist several designs for public verification even without the third-party auditor accessing the original data. Although this could be implemented into the design, it is out of scope since our objective is self-emerging PoS challenges where its design could be incorporated into a more complex and complete system.

Release-Ahead Attacks

The system is designed so that peers can register to the protocol at any time. Once they have registered, they can be selected to participate in a service for self-emerging PoS challenges by data owners. This means that peers cannot register to the system to participate in the service of a data owner after the data owner has set up the service. Consequently, no peer can target a specific data owner for an attack. This analysis is only valid in the case where several data owners and data storers utilize the system.

In a system where there only exists a single data storer, it could be beneficial for the data storer to control the peers in order to gain challenges before their release time. Controlling the majority of peers could be done by creating several new accounts and using them to register to the system as a peer. If a data storer is the only data storer in the system, and it controls the majority of the peers, it will be effortless to execute release-ahead attacks that cannot be detected. The gain of these attacks is, however, only to receive challenges before their release time. In a system where there exist several data storers, the likelihood that a data storer's controlled peers get selected for self-emerging challenges targeted to itself is small. To execute a release-attack successfully, the data storer would need to either control the last peer in a path or all. Even if it controls all peers in a path of a data owner's service, it still will not control the other potential paths in the service. Additionally, the reception of challenges before their release time is to little use to a malicious or faulty data storer that has not maintained the integrity of its clients' data. This is because the data storer still needs to prove the integrity of data by responding to the challenge. Based on the arguments presented, in a system where several data owners and data storers participate, it will be infeasible for a data storer to execute a release-ahead attack by controlling the peers.

Since controlling a service's selected peers in a path is infeasible given an ample system with multiple partaking data storers, the better choice of executing a release-ahead attack by a data storer is to bribe the last peer in a path to execute the attack on its behalf. However, the question of how much value this is to a data storer arises again. We have introduced that the reporter of a release-ahead attack is rewarded with $a = d^s$. This means that the data storer is awarded if it receives a challenge before its release time and reports the incident. Therefore, a bribed peer cannot be sure that the briber will not report it for executing a release-ahead attack.

Drop Attacks

Another interesting evaluation to expound around data storers is drop attacks. There is a value in receiving and correctly responding to a challenge. For a data owner, this value is ensuring its data integrity. For a data storer, this value is the gain of reputation that it gains for proving the integrity of the data. A data storer may, however, execute a drop attack if it possesses the knowledge that the integrity of data is not intact. The question is if not receiving a challenge has the same value to the data storer as correctly responding to a challenge. In the design, it is assumed that these do not have the same value. The basis of this assumption is that a data storer, in a system among several data storers, which constantly does not receive challenges earns a reputation just as negative compared to if it does not respond correctly to challenges. This especially applies when other data storers are constantly correctly answering challenges. Therefore, the most rewarding strategy of a rational data storer is always correctly respond to challenges without executing drop attacks.

Utilization of One Peer per Path

In Section 5.1 and 5.2, we showed that total gas costs in the system are reduced and that remuneration becomes more fair when only one peer is utilized per path. In this section, we will evaluate how the security of the system is affected by constantly allowing only one peer to deliver a challenge.

Peers pose a danger for the possibility of drop attacks. That is, each peer in a path has the possibility of executing a drop attack and render the service in that particular path useless. In the case where paths only contain one peer, the likelihood of drop attacks happening in a path is reduced since the number of peers is held at an absolute minimum. However, this increases the danger of a malicious data storer to control the paths in a system. This is because controlling a path requires less effort as controlling one peer equals to controlling the entire path. However, as discussed previously, a data storer that acquires a challenge before its release time has limited benefit as the data storer still needs to provide a correct response to the challenge in order to prove data integrity. Therefore, we conclude that the gain of fair remuneration, reduced gas costs and less likelihood of drop attacks outweigh the partially reduced security against release-ahead attacks and recommend that all paths should each contain one peer.

5.5 Future Direction

In this section, we present future directions related to our design that can be interesting to investigate. In addition, we provide some improvements that can be done to the existing solution.

Gas Optimization

One of the most important parts of evaluating an Ethereum smart contract is measuring the gas costs of functions that include the transfer of data or currency. Therefore, as a future work, it would be advantageous to analyze the design of the contract and optimize potential parts in order to reduce gas costs. Doing this results in a cheaper system for the data owners to issue self-emerging PoS challenges to their storage service providers.

Multiple Pairs of Data Owners and Data Storers in SEC Contract

As previously discussed, the design of the system assumes for security reasons that multiple data owners and data storers may form pairs. However, since the purpose of implementation has been to do an experimental evaluation of the proposed design, only the essential parts of the design are implemented in the SEC contract. The implemented solution only supports one pair of data owner and data storer. However, in Figure 5.5, we show how the system could be implemented to support multiple pairs. A smart contract can be made for peer registration and keeping multiple instances of child contracts. This contract contains a pool of registered peers. Data owners may set up a new service by sending a request to the contract. The contract creates a new instance of a child contract upon receiving such a request. Every operation related to a service will be executed in its own child contract, while still allowing data owners and data storers to be part of other child contracts as well. Thus, the logic of peer registration is separated from single services, and multiple pairs of data owners and data storers select peers from the same pool. We expect that creating such a parent contract to extract the peer registration logic into it does not increase the costs of the system. This is because the same amount of data is transferred and stored to the blockchain as with the current implementation. Therefore, we expect that no negative impact on costs will occur.

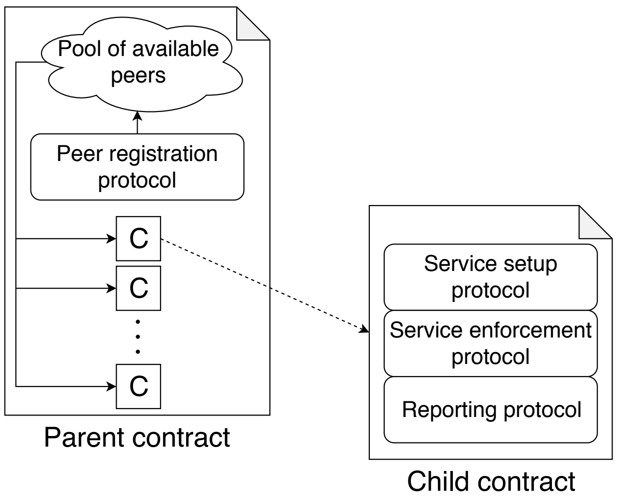

Figure 5.5: Structure of smart contracts for allowing multiple pairs of data owners and data storers as required by the design.

Establishing New Paths After Setup of Service

It is conceivable that a data owner may wish to issue new challenges to its data storer after setup of a service. The proposed design does not allow data owners to set up new paths for self-emerging challenges after the setup of a service. Allowing to do this requires extensive changes to the design. Currently, this can be done by creating a new service. That is, signing up for a new service which effectively is the same as creating new paths after the setup. However, as discussed in Section 5.1, setting up a new service is costly. Therefore, it could be advantageous to develop the design to include setup of new paths after the setup of the service and evaluate if gas costs could be reduced by having such a functionality instead of the current solution to the problem.

Incorporation of the Design into a Decentralized Storage System

The design of our proposed system is based on the use of smart contracts. Therefore, directly porting our design to a decentralized storage system for autonomous delivery of challenges and verification of proofs is not a trivial task if the system does not build on smart contracts. It would be interesting to investigate how the design could be incorporated into a decentralized storage system as a built-in part of a PoS protocol. Whether this decentralized storage system would be built on blockchain technology, or as its own platform would also affect how the transaction of currency would function. In case transactions could not be part of the system, the design would have to account for the security differently than the reward and penalty scheme that it currently possesses.

Addresses and Identities

The current design assumes that the identity of the data owner and data storer is known by each other when signing and pairing up to the system. The identity is given by the Ethereum account address. If the design is used as an extension of centralized storage services, the Ethereum addresses of data storers could be published on their respective web sites in plain text or as scannable QR-code. Furthermore, data owners would need to couple their Ethereum identity to their accounts in the data storers' systems. This could be done by submitting their Ethereum account addresses to their respective data storers. In a scenario where the system is used as an extension of centralized storage systems, algorithms for extracting the data storers that have the best ratio of correctly answered challenges and received challenges could be implemented since that data is publicly available in the blockchain. However, this may not be directly applicable in the case the proposed design is incorporated into a decentralized storage system. In such a system, the stored data is distributed amongst several nodes. If the data owners do not choose which nodes that store their data, they cannot not choose the data storers that have the best ratio of correctly answered challenges. Therefore, extensive changes to the proposed design need to be done to be able to achieve this. For example, a rating mechanism can be built into the decentralized storage system to avoid data storers that have a low ratio of correctly answered challenges.

Chapter 6

Conclusion

In this thesis, we presented a design for the automatic delivery of multiple challenges with definite time intervals using smart contracts. The proposed design offers functionality for self-emerging challenges at the data storer, and automatically verifies the data storers response to the challenges. The design is built on existing related work, namely the timed-release protocol and we solved the challenges related to incorporating this work into our design. The challenges were mainly related to changing the purpose of the self-emerging data scheme and changing the roles of the participants in the system. We showed that the differences required us to redesign vital parts of the timed-release protocol and extend it with additional functionality. Moreover, since the specifications of the timed-release protocol contained no detailed description of the smart contract in neither actual implementation nor pseudo code, we implemented the system from the bottom up and presented this work as a set of algorithms. Aside from the design of the smart contract, we presented a design for a module for utilizing the Whisper protocol as P2P communication through this protocol is a vital part of our design. Lastly, we introduced the peer selection algorithm and remuneration calculation that is a part of the timed-release protocol with the improvements and extensions in order to utilize them in our design.

We created a test setup for experimentation of the implemented system. This included the configuration of a private, local Ethereum blockchain network that enabled the Whisper protocol. Various, relevant parts of the system were tested, including the Whisper protocol module for P2P communication. Furthermore, the implemented smart contract that employs our design was deployed, and we tested each function in different service configurations to obtain gas costs.

We did an in-depth evaluation of the gas usage of the implemented Ethereum smart contract that employs our proposed design. Moreover, we analyzed in detail the security of the design and proposed mechanisms to prevent rational participants in the system from performing attacks. We also presented a comparison of the remuneration paid to participating peers for their service and acquired risk versus alternative methods of investing their deposits. By comparison, we showed that the system rewards peers for their service and acquired risk accordingly. Lastly, we discussed future directions regarding the total expenses of the data owner, the expansion to the architecture and design of the smart contract.

Through evaluations, we showed that total gas costs in the system are reduced and that fairness of remuneration is increased when exactly one peer is utilized in each path. By analyzing the affected security of always selecting one peer in each path, we discussed that the security against drop attacks are increased, while the security against release-ahead attacks are to some degree decreased. Since the benefits of utilizing exactly one peer per path outweigh the disadvantages, we conclude that data owners should always choose this approach when selecting peers for their services.

List of Figures

List of Tables

List of Algorithms

List of Listings

Appendix A

Implementation and Test Input Data

The implementation of the smart contract, local Ethereum network configuration, Whisper module and the raw test input data that has been used for obtaining the gas costs of the system is published in the following Github repository: `https://github.com/danielbarati/sec`.

A private, local Ethereum network can be deployed by running the configuration with a working installation of Docker.

The implemented smart contract can be deployed on an Ethereum network, using Remix IDE [27] or using Ganache [14].

Appendix B

Complete Experimental Data

This appendix contains tables of gas costs of the implemented smart contract. The test results are presented for different configurations of the system. Both the number of paths and the number of peers in each path is varied. Cells marked with '-' denote the function irrelevant for the particular participant. E.g. since the `newPeer` function is only relevant for peers; it is marked with '-' in the cell for the data owner. Furthermore, for cases where the system contains multiple paths, the gas cost for the actions that the data storer needs to perform are divided into multiple columns. Cells marked with '*' in these columns denotes that the particular function is only invoked once and therefore is not relevant for multiple paths. The gas costs of functions that are only invoked once are stated in the column for the first path of the data storer.

The measurements of gas costs given in the following tables can be obtained by deploying the implemented smart contract and invoking the functions using the test input data. Instructions of this are given in Appendix A. The test results are obtained by using deployed contract compiled with Solidity compiler (0.5.1+commit.c8a2cb62.Emscripten.clang) and the Javascript VM in Remix [27]. In all tests, the public keys have a 2048 bits key size and all plain text challenges, answers and certificates are 16 bits.

Ethereum Gas Cost			
Function	Owner	Peer	Storer
newPeer	-	324 998	-
updatePubKey	-	81 307	-
updateBalance	-	27 580	-
updateWindow	-	32 779	-
ownerSign	85 473	-	-
storerSign	-	-	241 922
setup	279 358	-	-
setProps	219 686	-	-
verifyCert	-	44 633	25 583
setWhisperKey	-	44 805	-
verification	-	29 749	-
answer	-	-	75 959
releaseReport	-	-	85 938
dropReport	-	98 217	91 264

Table B.1: Gas cost of each function in the smart contract. Configuration with one peer in one path.

Ethereum Gas Cost				
Function	Owner	Peer 1	Peer 2	Storer
newPeer	-	325 062	311 014	-
ownerSign	85 473	-	-	-
storerSign	-	-	-	241 922
setup	369 334	-	-	-
setProps	221 368	-	-	-
verifyCert	-	44 633	45 520	25 919
setWhisperKey	-	44 805	45 628	-
verification	-	29 749	30 572	-
answer	-	-	-	75 184
releaseReport	-	-	-	85 623
dropReport	-	97 904	125 633	91 750

Table B.2: Gas cost of functions in the smart contract except peers' update functions. Configuration with two peers in one path.

	Ethereum Gas Cost				
Function	Owner	Peer 1	Peer 2	Peer 3	Storer
newPeer	-	325 062	311 014	311 838	-
ownerSign	85 473	-	-	-	-
storerSign	-	-	-	-	241 922
setup	459 374	-	-	-	-
setProps	244 366	-	-	-	-
verifyCert	-	44 633	45 520	46 078	26 742
setWhisperKey	-	44 805	45 628	46 451	-
verification	-	29 749	30 572	31 395	-
answer	-	-	-	-	74 409
releaseReport	-	-	-	-	85 286
dropReport	-	97 567	125 734	127 536	92 236

Table B.3: Gas cost of functions in the smart contract except peers' update functions. Configuration with three peers in one path.

	Ethereum Gas Cost				
Function	Owner	Peer 1	Peer 2	Storer path 1	Storer path 2
newPeer	-	325 062	311 014	-	-
ownerSign	85 473	-	-	-	-
storerSign	-	-	-	241 922	*
setup	412 447	-	-	-	-
setProps	331 628	-	-	-	-
verifyCert	-	44 633	45 520	26406	26470
setWhisperKey	-	44 805	45 628	-	-
verification	-	29 749	30 686	-	-
answer	-	-	-	75 959	46 223
releaseReport	-	-	-	85 938	57 560
dropReport	-	98 219	70 664	94 234	67 314

Table B.4: Gas cost of functions in the smart contract except peers' update functions. Configuration with two paths, each containing one peer.

Ethereum Gas Cost							
Function	Owner	Peer 1	Peer 2	Peer 3	Storer path 1	Storer path 2	Storer path 3
newPeer	-	325 062	311 014	311 838	-	-	-
ownerSign	85 473	-	-	-	-	-	-
storerSign	-	-	-	-	241 922	*	*
setup	442 693	-	-	-	-	-	-
setProps	443 571	-	-	-	-	-	-
verifyCert	-	44 633	45 520	46 343	27229	27293	27293
setWhisperKey	-	44 805	45 628	46 451	-	-	-
verification	-	29 749	30 686	31 509	-	-	-
answer	-	-	-	-	75959	46223	46223
releaseReport	-	-	-	-	85938	57560	59018
dropReport	-	98 219	70 664	74 215	92910	64532	65990

Table B.5: Gas cost of functions in the smart contract except peers' update functions. Configuration with three paths, each containing one peer.

Ethereum Gas Cost	
Function	Owner
setup	855 655
setProps	581 047

Table B.6: Gas cost of data owner setup for a configuration with three paths, each containing three peers.

Ethereum Gas Cost			
Function	512 bit public key	1024 bit public key	2048 bit public key
newPeer	178 325	222 005	325 062

Table B.7: Comparison of peer registration gas cost using different sized public keys.

Bibliography

[1] Marketwatch. Cloud Storage Market to 2024 Trends, Competitive Analysis and Growth Opportunity. https://www.marketwatch.com/press-release/cloud-storage-market-to-2024-trends-competitive-analysis-and-growth-opportunity-2019-01-16, January 2019. Accessed: 2019-06-10.

[2] Veerawali Behal and Rydhm Beri. Cloud Computing: A Survey on Service Providers. 2, 03 2016.

[3] StorageCraft. Which Hardware Fails the Most and Why. https://blog.storagecraft.com/hardware-failure/. Accessed: 2019-06-10.

[4] Seny Kamara. Proofs of storage: Theory, constructions and applications. In Traian Muntean, Dimitrios Poulakis, and Robert Rolland, editors, *Algebraic Informatics*, pages 7–8, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Chao Li and Palanisamy Balaji. Decentralized Release of Self-emerging Data using Smart Contracts. In *37th IEEE International Symposium on Reliable Distributed Systems*, 2018. URL http://d-scholarship.pitt.edu/34983/.

[6] Viktor Tron, Aron Fischer, and Nick Johnson. smash-proof: auditable storage in swarm secured by masked audit secret hash. 2016. URL http://swarm-gateways.net/bzz:/swarm/ethersphere/orange-papers/2/smash.pdf. Ethersphere Orange Papers 2.

[7] Protocol Labs. Filecoin: A Decentralized Storage Network. July 2017.

[8] Juan Benet, David Dalrymple, and Nicola Greco. Proof of Replication Technical Report (WIP). 2018.

[9] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In Mitsuru Matsui, editor, *Advances in*

Cryptology – ASIACRYPT 2009, pages 319–333, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10366-7.

[10] Ethereum. A Secure Decentralized Generalized Transaction Ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`, 05 2019. Accessed: 2019-06-10.

[11] Etherchain. Average Block Time of the Ethereum Network. `https://www.etherchain.org/charts/blockTime`. Accessed: 2019-06-10.

[12] Kevin Leyton-Brown and Yoav Shoham. *Essentials of Game Theory: A Concise Multidisciplinary Introduction*, volume 2. 01 2008. ISBN 978-1598295931. doi: 10.2200/S00108ED1V01Y200802AIM003.

[13] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950. ISSN 0027-8424. doi: 10.1073/pnas.36.1.48. URL `https://www.pnas.org/content/36/1/48`.

[14] Ganache. One Click Blockchain. `https://truffleframework.com/ganache`. Accessed: 2019-06-10.

[15] Rinkeby. An Open-Source, Public, Ethereum Platform. `https://www.rinkeby.io/#stats`. Accessed: 2019-06-10.

[16] Github. Ganache-CLI. `https://github.com/trufflesuite/ganache-cli`. Commit: 4f57d6a0e4c030202a07a60bc1bb1ed1544bf679. Accessed: 2019-06-10.

[17] Geth. Official Go Implementation of the Ethereum Protocol. `https://github.com/ethereum/go-ethereum/`. Accessed: 2019-06-10.

[18] Docker. Software for Operating-System-Level Virtualization. `https://www.docker.com`. Accessed: 2019-06-10.

[19] Bitbucket. The Ethereum-Block-Explorer. `https://bitbucket.org/designisdead/blockchain-explorer.git`. Accessed: 2019-06-10.

[20] Truffle Suite. Tools For Smart Contracts. `https://www.trufflesuite.com`. Accessed: 2019-06-10.

[21] Etherscan. Ethereum Average Gas Price Chart. `https://etherscan.io/chart/gasprice`, . Accessed: 2019-04-29.

[22] Etherscan. Block Explorer, Search, API and Analytics Platform for Ethereum. `https://etherscan.io/chart/etherprice`, . Accessed: 2019-04-29.

[23] Investopedia. What Is the Average Annual Return for the S&P 500. `https://www.investopedia.com/ask/answers/042415/what-average-annual-return-sp-500.asp`, May 2019. Accessed: 2019-06-10.

[24] Dropbox. Choose the Right Dropbox for You. `https://www.dropbox.com/plans`. Accessed: 2019-06-10.

[25] Statista. Price of Ethereum from April 2017 to April 2019 (In U.S. Dollars). `https://www.statista.com/statistics/806453/price-of-ethereum/`.

[26] The Telegraph. Understanding Cryptocurrency Market Fluctuations. `https://www.telegraph.co.uk/business/business-reporter/cryptocurrency-market-fluctuations/`, July 2018. Accessed: 2019-06-10.

[27] Remix IDE. Open Source Tool for Writing and Testing Ethereum Smart Contracts. `https://remix.ethereum.org/`. Accessed: 2019-06-10.