A Community-run Decentralised Loan Liquidation Scheme Based on an Ethereum-enabled DeFi System

Jude Molloy

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Liquidating loans on decentralised lending protocols requires significant domain knowledge, preventing most decentralised finance users from participating in this vital process. A pool-based protocol can provide users with access to this process regardless of their technical expertise. The protocol - operated and governed by users who do not necessarily trust each other - utilises an off-chain oracle to discover liquidation opportunities and trigger liquidations on-chain, earning a profit for the protocol participants. The proposed scheme consists of DeFi users depositing their funds into a community-governed and managed smart contract that will liquidate unhealthy lending positions on the Aave protocol based on the identification of a liquidation opportunity by a community-delegated off-chain program that searches the blockchain for unhealthy lending positions. Loan liquidations are a relatively secretive aspect of Ethereum development (since they can be highly profitable) and so it is a challenge to find useful information on how to perform them effectively. The scheme proposed in this paper is a gas-efficient and secure system that facilitates this functionality, exploring the design choices that must be made to create a community-run system in the context of decentralised finance. Whilst the development of this scheme was a success, there is still a large scope to expand on this work, particularly in the efficient identification of liquidation opportunities.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jude Molloy*)

Acknowledgements

I would like to thank my honours project supervisor Xiao Chen for his support this year.

Also, I would like to thank my family and friends, particularly my parents for their unfaltering support throughout my time at university.

Table of Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Objectives . . . 3

2 Background **4**
- 2.1 Ethereum . . . 4
- 2.2 Aave: A Decentralised Lending Protocol . . . 5
- 2.3 Chainlink: A Decentralised Data Feed . . . 7
- 2.4 Uniswap: A Decentralised Exchange (DEX) . . . 7
- 2.5 The Graph: A Decentralised Indexing Protocol . . . 10
- 2.6 Related Work . . . 10

3 Design and Implementation of Decentralised Liquidation System **12**
- 3.1 General System Requirements . . . 12
- 3.2 Liquidation Opportunity Oracle (LOO) . . . 13
- 3.3 Community Pool Protocol (CPP) . . . 15
  - 3.3.1 Executing the Loan Liquidation . . . 15
  - 3.3.2 Smart Contract Architecture . . . 16
  - 3.3.3 Governance Abilities . . . 23

4 Experiments and Evaluation **29**
- 4.1 Experimental Setup . . . 29
  - 4.1.1 Experimental Environment . . . 29
  - 4.1.2 Simulating a Price Drop . . . 30
  - 4.1.3 Testing the Liquidation Call . . . 30
- 4.2 Latency of the executeLiquidation Function . . . 31
- 4.3 Gas Analysis . . . 32
  - 4.3.1 Early Exit Require Statements . . . 32
  - 4.3.2 Data Structure and Variable Usage . . . 33
  - 4.3.3 Further Gas Efficiency Techniques . . . 34
  - 4.3.4 Gas Values for CPP Smart Contract Functions . . . 35
  - 4.3.5 Gas Efficiency's Impact on Profitability . . . 36
- 4.4 Security and Decentralisation Evaluation . . . 37

5 Conclusion and Further Work **39**

Bibliography **41**

Chapter 1

Introduction

1.1 Context and Motivation

After the advent of crypto-tokens on the Ethereum blockchain, capital poured into the ecosystem and just like the assets of the traditional finance world, these cryptocurrency tokens necessitated corresponding financial infrastructure.

In the same manner as lending in traditional finance (TradFi), decentralised lending is a major aspect of decentralised finance (DeFi) on the Ethereum blockchain. Whether it be for accessing liquidity without having to sell their current holdings or to construct some trading position, decentralised lending protocols provide decentralised finance users with access to similar financial tools to the traditional finance world in a permissionless fashion.

Many decentralised lending protocols exist, though, this paper focuses on Aave [16] - the largest lending protocol on the Ethereum blockchain by market capitalisation - which had more than 18 billion US dollars in its lending pools at its peak on the 29th of October 2021 [13]. Loans on Aave are overcollateralised - meaning that the collateral that a user has must be more than what they are borrowing - and a user's lending position condition is measured by what is called a "health factor". When the "health factor" of a loan falls below 1, it is deemed too risky and it must be liquidated. Liquidation of a loan is the process of selling assets or collateral pledged by the borrower to recover the outstanding loan amount when the borrower is unable to repay the loan.

In traditional finance, loans are managed by a bank that ensures they are sufficiently collateralised and healthy, liquidating their customers' lending positions as they see fit. Aave is a decentralised protocol with no intermediary between borrowers and lenders, hence, the loans are liquidated in a decentralised manner by incentivising third parties to liquidate unhealthy loans via a liquidation bonus.

Whilst liquidating these loans can be lucrative, doing so requires a considerable understanding of the underlying protocol as well as the technical ability to produce an effective liquidation system. Liquidators must understand how loans work on the protocol as well as carry out the process of actually executing the liquidation on-chain which can prove to be a costly and not profitable process if performed incorrectly.

Due to this, the rewards earned via the liquidation process are captured by a small portion of the market with esoteric expertise. This is undesirable since one of the core principles of blockchain technology is decentralisation and the degree of decentralisation of the resources - in Ethereum's case the crypto assets and therefore the liquidation rewards - is the dominant metric for assessing the level of decentralisation within a blockchain [22].

Research carried out by Qin et al. on DeFi liquidations confirms that only a small percentage of the market is earning from the liquidation process with 1,039 Aave V2 loans being liquidated by only 125 unique liquidators over the two-year period of their study. The overall findings showed that the $63.59 million profit earned via liquidations over the period was spread among only 2,011 unique liquidators all making $31.62K on average with the most profitable liquidator making $5.84 million in profits from liquidations [28]. Considering that this study uses data from a few years ago, it is not a mental leap to suggest that the profits earned via liquidations now are far larger due to the increased capital deposited in such protocols.

Furthermore, partitioned liquidations are less economically efficient due to the gas fees incurred from multiple liquidation calls rather than just one. Facilitating DeFi users without sufficient capital to individually liquidate the loan to pool their resources with others in the same position to collectively liquidate the loan, each paying a fraction of the gas fee that they would have. This also allows DeFi users to allocate smaller amounts of capital to the system that otherwise would prove to be pointless due to fees.

A trustless, pool-based system can address these problems, democratising access to loan liquidations in a decentralised way. Users can come together, pooling their funds to liquidate unhealthy loans on the Aave protocol. They can participate regardless of their wealth and understanding, depositing an amount of their choosing and can treat the system like a black box.

The proposed scheme is comprised of two main systems, the Liquidation Opportunity Oracle (LOO) and the Community Pool Protocol (CPP). The LOO - which operates off-chain - searches the blockchain for unhealthy loan positions that can be liquidated. It does this by aggregating all of the lending positions on the Aave protocol and reconstructing their health factor, identifying users with health factors below 1 since this means that they have collateral that can be acquired at a discount by liquidating their debt position. Once an unhealthy lending position has been found, this information is sent to the CPP which handles the execution of the liquidation on-chain and enables the decentralised governance of the protocol. The CPP is a smart contract which facilitates the on-chain aspect of the system, allowing users to deposit funds to be used for liquidations. The smart contract carries out the loan liquidation on Aave based on information sent to it from the LOO which it itself verifies on-chain. The CPP smart contract also enables the system to be governed proportionately by the participants of the pool. This helps ensure the integrity of the system and allows the system to be malleable in adverse situations or even just changes in the directional beliefs of the stakeholders of the system.

The democratisation of the liquidation process, promoting decentralisation of earnings and therefore decentralisation of the whole Ethereum blockchain, heavily motivates the

work of this paper in an attempt to advance the vision of blockchain enthusiasts who anticipate a decentralised future.

1.2 Objectives

The main objectives for this project are to propose a design for a system that:

- Allows DeFi users to collaborate in a trustless manner to liquidate loans on the Aave protocol by pooling their funds.
- Operates in a decentralised manner, creating solutions to the trade-off in the level of centralisation that is required for the system to run efficiently.
- Explores and justifies design choices relating to the system by considering the context of decentralised finance and detailing the best methods to:
    - Identify unhealthy user lending positions.
    - Determine whether a liquidation opportunity is likely profitable or not.
    - Manage users' deposited funds.
    - Avoid unnecessary risk.
    - Help stakeholders govern the system in a decentralised manner.
- Implements a secure and gas-efficient smart contract design.
- In a more general sense demonstrates how systems can be designed to enable DeFi users to collaborate in a trustless manner when aspects of centralisation are required.

Chapter 2

Background

2.1 Ethereum

Ethereum is an open-source blockchain that runs on a decentralised network of computers, making it more resistant to censorship and downtime than a centralised network. The term, "smart contract" is heavily associated with the Ethereum blockchain, though, this is actually a misnomer [21] as they are not necessarily "smart" or "contracts", rather, they are just programs that run on the Ethereum network.

The state, in Ethereum, is made up of objects known as "accounts". There are two types of accounts [11]:

- Externally Owned Accounts (EOA)
    - Managed by a private key.
    - Tends to be operated by an individual.
    - Does not have code associated with it.
- Contract Accounts
    - Managed by smart contract code.
    - Can execute code.

Smart contracts can be used to develop things like decentralised applications (dApps) - such as the decentralised finance applications discussed in this paper - and cryptocurrency tokens. These cryptocurrency tokens differ from the native currency of the Ethereum blockchain which is called ether (ETH) and is used to pay for transaction fees on the network and as a store of value. The cryptocurrency tokens are fundamentally just smart contracts that follow the ERC-20 standard [27] which sets rules and functionality that allow the tokens to be compatible with the Ethereum ecosystem allowing Ethereum accounts to own, store and transfer these tokens between other accounts.

ERC-20 tokens are extremely important on Ethereum, representing various types of assets, from highly volatile speculative tokens to stablecoins pegged to the US Dollar or even tokens representing ownership of an organisation or protocol.

The rise of these tokens meant that capital flowed into the Ethereum ecosystem requiring the financial infrastructure to allow users to fully utilise the tokens whether it be lending, trading or insurance.

2.2 Aave: A Decentralised Lending Protocol

Aave is a decentralised liquidity protocol built on the Ethereum blockchain that allows users to borrow and lend various cryptocurrencies in a trustless, open-source and non-custodial manner; no intermediary party is required to facilitate the loans.

This system is built around lending pools which allow users to pool their funds so that other users can then borrow from these lending pools at an interest rate that is algorithmically determined based on the supply and demand of a given pooled asset. Lending pools are technically implemented as smart contracts. Borrowing assets on the Aave protocol is perpetual with no repayment schedule; partial and full repayments of borrows can be made at any time.

Users are incentivised to lend their money on the platform by depositing their assets to the pool since they will earn interest by doing so. On the other end of this, a borrower may withdraw the asset from the pool, paying interest.

Aave has two types of interest rates: a stable rate and a variable rate. The stable rate is fixed in the short term, although, can be rebalanced in the long run depending on the condition of the market. This will appeal to users that wish to have a stronger idea about the amount of interest they will have to pay on their loan, though, the variable rate may end up being the optimal rate over a period of time. The variable rate is the rate that is determined based on the supply and demand for a lending pool as well as its collateralisation ratio - the ratio of the value of the collateral to the value of the loan.

Aave makes use of aTokens in its protocol, which represents the balance of the underlying asset that a user has provided a lending pool with. When a user deposits an asset into a lending pool they receive an equivalent amount of aTokens in return. The number of aTokens that they own is continually increased in their wallet balance based on the yield earned by providing their capital to the lending pool.

Unlike loans in traditional finance, there is no strong credit rating system that lenders can use to determine how trustworthy a user is as a borrower. This is because users' accounts are not necessarily linked to a real-world identity. Since loans are not made based on a credit score, loans on the Aave protocol are overcollateralised i.e. the amount borrowed is less than the collateral that a user has. (Note: Flashloans on Aave are undercollateralised, though this is a separate mechanism and the utility of the borrowed funds is limited.) The value that can be borrowed using a specific asset as collateral is determined by its loan-to-value ratio (LTV) - which is expressed as a percentage - and is dependent on the perceived riskiness of the asset. For example, an asset with $LTV = 75\%$ would mean that the user could borrow up to 75% of the value of that specific collateral asset.

Only certain assets - determined by Aave's governance - can be used as collateral. Assets that are too volatile or are not deemed to be sufficiently decentralised are not

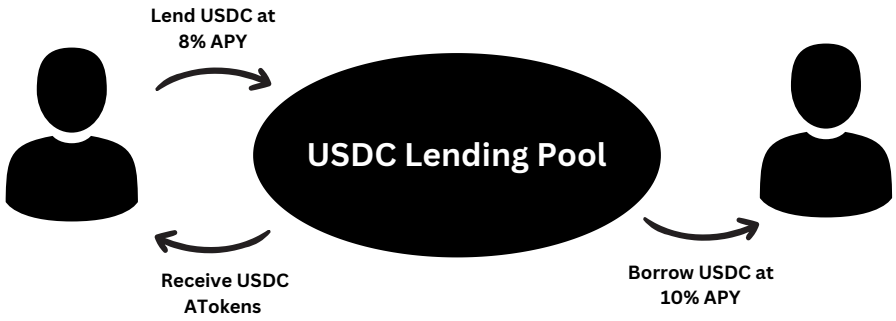

Figure 2.1: An example of a lending pool on the Aave protocol. User 1 (left) deposits USDC into the lending pool which offers an interest rate of 8% APY and in return receives the interest-bearing USDC ATokens that represent their deposit to the lending pool. User 2 (right) can borrow USDC from this lending pool (provided they have sufficient collateral) at an interest rate of 10% APY.

used as collateral as their risk could have an adverse impact on the overall health of the protocol.

To ensure the protocol remains solvent the loan positions must be managed, liquidating any loans that become sufficiently undercollateralised. Thus, assessing the state of a loan is an essential process of the protocol and is measured on Aave using what is called a health factor. The health factor of a loan position is defined by [5] :

$$H_f = \frac{\Sigma_i \ (Collateral_i \ in \ ETH \times Liquidation \ Threshold_i)}{Total \ Borrows \ in \ ETH} \tag{2.1}$$

- Where i is an asset that the user has deposited to a lending pool.
- Note: Both the collateral and borrow amount are valued in ETH.

The health factor is equal to the sum of the value of each collateral asset that the user has deposited multiplied by the asset's respective liquidation threshold all divided by the total amount borrowed. Essentially, the health factor expresses the combined health of all lending positions that a user currently has on the protocol.

The liquidation threshold, which varies by asset dependent on its associated risk, is defined as the percentage at which a position is considered undercollateralised and could be liquidated. The liquidation threshold is similar to LTV, though, it is often larger by a few per cent. The difference between an asset's LTV and its liquidation threshold provides the borrower with some protection.

When a borrower's health factor, H_f, falls below 1, up to 50% of their debt is available to be liquidated to bring their health factor back above 1. Aave is a decentralised protocol so there is no central controlling party that actively manages the health of these loans but loans still must be liquidated to ensure that the Aave protocol remains solvent. To carry this out in a decentralised manner, the protocol uses liquidation bonuses to incentivise DeFi market participants to carry out the liquidations. The liquidation bonus of an unhealthy loan is the percentage reward that is earned when liquidating a loan

and varies asset by asset. For example, a liquidation bonus value of 8% means that liquidating 10 ETH worth of debt would result in receiving 10.8 ETH worth of the collateral asset.

2.3 Chainlink: A Decentralised Data Feed

In many ways, the Ethereum blockchain is isolated from the world but many decentralised applications require reliable real-world data to operate correctly. Chainlink acts almost as an intermediary between the blockchain and the real world, allowing DApps to have access to definitive real-world data such as price feeds for financial market data including commodities, stocks, fiat currencies as well as cryptocurrencies by simply calling the functions of the Chainlink protocol's smart contracts. Chainlink provides a system that incentivises parties to provide this data reliably and in a decentralised manner, discouraging dishonest behaviour through economic incentives.

Real-time, accurate price data is extremely important, particularly in financial settings since the numbers being slightly off can lead to significant knock-on effects. For this reason, Chainlink is utilised throughout the design of the system as a reliable source of unbiased information.

2.4 Uniswap: A Decentralised Exchange (DEX)

An extremely important aspect of the DeFi ecosystem is the exchanges that allow cryptocurrencies to be traded similarly to the traditional financial (TradFi) markets. Uniswap is the largest decentralised exchange (DEX) on the Ethereum blockchain with around 10 billion dollars in total value locked at its peak in May 2021 [14]. Created in 2018 by Hayden Adams, Uniswap was the first decentralised exchange to make use of an automated market maker (AMM).

An automated market maker model enabled the exchange of various cryptocurrency tokens on the Ethereum blockchain in a way that is significantly different to how markets are made in traditional finance.

In TradFi, markets typically use an order-book system to manage trades. Though order books are generally the method used in centralised financial systems there have been some attempts made to bring this system on-chain, however, a system like this requires a high number of transactions per second (TPS) that most decentralised blockchains cannot currently facilitate. As well as this, the relatively low volume that is traded on decentralised order-book exchanges such as Serum means that large industry players cannot utilise the exchange effectively. In the order-book model, a list of buy and sell orders for specific assets are stored digitally in descending order by price. Whenever a buyer wishes to purchase an asset, they bid at a specific price. This price is then matched to a seller that will sell at the required price and the trade is executed - if there is a matching order. Similarly, when one wishes to sell an asset, they place an ask order at a specific price and again, this price is matched to an order in the order book (if a match exists) and the trade is executed. In both cases, if there is no matching order, the

bid/ask is added to the order book until a matching order is added. In this system, many large TradFi firms put their capital to work by becoming "market makers" providing liquidity to the market by placing orders on both sides of the book and earning the spread (the difference between the bid and ask prices). This helps ensure that there is always a buyer and seller for a given asset even when there are not necessarily enough buyers/sellers in the general market. They do this by employing various algorithms that determine the prices at which they are willing to buy and sell considering the market conditions. This is extremely difficult to do profitably and thus only large financial firms can perform it efficiently due to the large amount of data and quality of algorithms required.

DeFi is different. The AMM model allows users with any amount of capital and no significant expertise to participate in a similar market process by providing liquidity to the liquidity pools that AMMs utilise. These liquidity pools usually hold a pair of tokens (though there are some instances of multi-token liquidity pools). For example, for a USDC/WETH liquidity pool, liquidity providers provide an equal amount (in USD) of USDC and WETH. They are incentivised to do this as they can earn a proportion of the fees earned by the liquidity pool when users swap tokens. The proportion that they earn in fees is based on the proportion of the liquidity that they have provided to the pool. This process is not entirely risk-free though, since liquidity providers are exposed to the risks of impermanent loss - the loss in value of a liquidity position in comparison to holding the original assets as the price changes [18].

Instead of holding a list of orders and trying to match them, automated market makers use an invariant equation to determine the price at which an asset pair should be traded. The first and most basic type of AMM is a constant product market maker which calculates the exchange rate based on the following formula [7]:

$$x \cdot y = K \tag{2.2}$$

where x and y are the amounts of each token in the token pair's liquidity pool and K is the pool constant. The relationship of x and y can be plotted on a graph as shown in (Fig. 2.2).

This formula produces the following 'bonding curve' (Fig. 2.2) for the buy/sell price of the token pair. When a trade is made on Uniswap the price is updated based on the invariant formula above (2.2), not based on the price as determined by other external markets. When a user wishes to withdraw (buy) an amount of token x, they must deposit (sell) a proportional amount of token y to maintain the constant value K. The exchange rate for token x can be calculated using the following formula:

$$P_x = \frac{y}{x} \tag{2.3}$$

Similarly, the exchange rate for token y is calculated using:

$$P_y = \frac{x}{y} \tag{2.4}$$

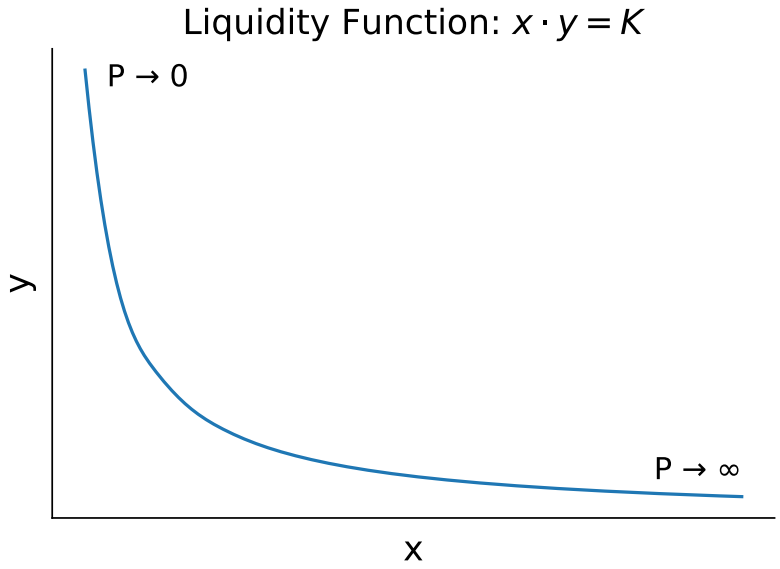

Figure 2.2: The bonding curve for the liquidity function: $x \cdot y = K$. The relative amounts of the two tokens x, y and the constant K determine the position on the curve and therefore the price.

A user attempting to purchase an amount Δy of tokens, hence reducing the liquidity in y, would have to input an amount Δx of tokens that satisfies the market-making functions constraint. The current amounts of tokens (x and y) in the pool the amounts after the purchase ($x + \Delta x$ and $y - \Delta y$) must satisfy equation 2.2 with their product equalling K. Hence, the following equation [7]:

$$x \cdot y = (x + \Delta x) \cdot (y - \Delta y) \tag{2.5}$$

Rearranging this equation, Δx is the amount of tokens that would need to be paid for Δy tokens,

$$\Delta x = \frac{\Delta y}{y - \Delta y} \cdot x \tag{2.6}$$

When using a DEX such as Uniswap, price impact must be considered. Price impact is the change in token price directly caused by your trade [20]. The price equations above (2.3 and 2.4) show the price for the marginal token, although often, trades are for many tokens at once. In this case, every token costs more than the previous one as the reserve amounts in the pool are altered and the price moves along the bonding curve [17]. This is described in equation 2.6 above; notice that as Δy is increased, the Δx that is required increases rapidly if Δy is more than just a small fraction of y. Hence, the size of a trade compared to the overall liquidity of the asset pair in the pool is what determines the magnitude of the price impact. Pools with high liquidity will have a lower price impact (unless the trade is very large) than trades in pools with low liquidity.

The price is kept at (or very close to) the actual market price by market participants through arbitrage. If the price on Uniswap is higher than the real market price then arbitrageurs are incentivised to purchase the token on another exchange at the market price and then sell it at the higher price on Uniswap, and vice versa. When this happens either from a single large arbitrageur or a group of smaller trades, the price is driven along the bonding curve closer towards the market price until the arbitrage is no longer profitable. Overall, this ensures that the price on the AMM closely tracks the true market price [8].

As the Uniswap protocol has developed over the years, it has made some optimisations concerning how it works. The Uniswap V3 protocol introduces the concept of concentrated liquidity allowing users to allocate their funds to a pool within a custom price range [6]. This means that a Uniswap V3 liquidity pool can operate with higher liquidity using fewer resources than a pool on a previous version of Uniswap.

Though the optimisations made in Uniswap V3 mitigate the price impact problem they certainly do not solve it, thus, it is still an important factor that must be considered before executing a trade.

2.5 The Graph: A Decentralised Indexing Protocol

The Graph [30] is a decentralised indexing protocol that allows decentralised applications (dApps) to query on-chain data simply and efficiently. It provides this data in the form of subgraphs which are structured representations of related data submitted to The Graph network. For example, Aave has its own subgraph which can be queried to fetch reliable data about various aspects of the Aave protocol such as the lending pools or the loan positions of users. Without a protocol like this, dApps would have to accumulate data manually by calling smart contract view functions and fetching historical data from emitted events on the blockchain. Through this system, reliable data is available in an efficient, well-documented and decentralised manner for dApps to use.

2.6 Related Work

The work of this paper continues and expands on a small paper written for the UKPEW 2022 Workshop [24]. The short paper briefly describes the motivations for such a system and gives a rough overview of how to implement the scheme. The work presented in this project delves much deeper, exploring the design choices further, implementing the system, and experimenting with and evaluating it.

Loan liquidations on decentralised lending protocols are a relatively new research area and therefore there is not much literature exploring how to implement systems that effectively liquidate decentralised loans and there is certainly nothing related to how to do this in a collective and decentralised manner. The work in this project presents a unique and original system to allow untrusting DeFi users to collaborate on the liquidation of loans. There is currently nothing like this on the market, nor is there any research exploring how a system like this would work. Whilst loan liquidation bots do

exist (just private programs that solely carry out the liquidation part), none are known to exist that work in a community-run manner allowing for the layperson to participate and earn from the process.

Most of those who understand decentralised loan liquidations are generally unwilling to publish information about it since superior knowledge on the subject is a competitive advantage and in the nascent markets, this can lead to substantial profits. The single entity-run liquidation bots that do exist are not yet covered in academic literature or the general web. The systems that people develop and the design decisions that they make are secret knowledge that they're unwilling to share. This work navigates some of the decisions that are required when developing such systems and explains the reasoning behind these design decisions in the decentralised finance context.

Whilst this secretive nature is unfortunate, it leaves a lot of avenues for ingenuity when working with decentralised loan liquidations as there is no "standard" way to do things; design choices are still being explored.

Though there is not any directly related work, there does exist some research in the general space of decentralised lending. One particularly impactful paper would be that of Qin et al. [28] which researched various decentralised lending protocols on the Ethereum blockchain such as Aave, MakerDAO, Compound and dYdX. The research acquired data from all of the protocols which it presented, this data and exploration helped inform the work of this project by providing insight into the general liquidation activity within the Ethereum blockchain. That being said, their work is more concerned with the efficiency and fairness of the liquidation mechanisms used by these protocols. This focus is shared by another influential paper in the area by Perez et al. [26] which investigates liquidation efficiency on Compound.

Furthermore, another important related paper is the work carried out by Bartoletti et al. [9] which offers a formal model of lending pools, describing their interactions. This paper was useful in understanding the DeFi lending systems allowing the work of this project to make informed design decisions.

Chapter 3

Design and Implementation of Decentralised Liquidation System

In this chapter, the design and implementation of the decentralised liquidation scheme are explored. The system consists of two main components:

- The Liquidation Opportunity Oracle (LOO).
- The Community Pool Protocol (CPP).

These two components combine to make the whole system, though, the majority of the system relies on the Community Pool Protocol which runs on the Ethereum blockchain. Throughout this chapter, the general requirements of the system are considered, the LOO design is described and justified, and the architecture and reason for the design decisions of the CPP smart contract are detailed. Finally, the design and implementation of the governance abilities of the system are presented.

3.1 General System Requirements

Presented below is a brief overview of the general requirements that the system must satisfy. The system must:

- Periodically fetch and store data about users' lending positions on the Aave protocol.
- Calculate the health factor of users' lending positions to identify those that can be liquidated.
- Call the smart contract, from an off-chain program, with the relevant data to execute the loan liquidation.
- Allow users to pool their funds in a decentralised and trustless manner.
- Validate, on-chain, that an identified loan is likely to be a profitable liquidation opportunity.

- Acquire the debt asset through a decentralised exchange so that it can be used to execute the liquidation.
- Execute the liquidation on the Aave protocol.
- Sell the collateral asset at its market price via a decentralised exchange.
- Calculate and distribute loan liquidation profit.
- Facilitate governance of the system allocating voting power proportionately to each participant's stake in the system.

3.2 Liquidation Opportunity Oracle (LOO)

To liquidate a loan, the account that has an unhealthy position in the protocol must be known. Such accounts can be found via two main methods; either by looking directly at on-chain data, namely, the emitted events of the Aave protocol or by using GraphQL to query Aave's subgraph on The Graph [30] (an indexing protocol for querying the Ethereum network). The proposed system uses The Graph to fetch on-chain data.

There are a few reasons that The Graph is the method of choice for the system. The benefits to using The Graph rather than working directly with emitted events are:

- The Graph offers a scalable solution for fetching large amounts of data efficiently, whereas, emitted events can be slow and resource intensive.
- The Graph is a developer-friendly protocol that uses a GraphQL style API, making it simple to query and index data. This reduces implementation time and prevents the code base from becoming cumbersome to work with.
- Aave has created its own subgraph, optimising it, so that it makes for a suitable method to fetch the data that is needed.
- Data standardisation on The Graph is important since it allows for easy access compared with data that is potentially partitioned into many confusing types of emitted events.

The LOO system periodically makes requests to Aave's subgraph on The Graph to fetch the data required to reconstruct the lending positions of users. The periodicity of the calls is a topic that requires dedicated research to determine the optimum frequency at which to fetch data efficiently.

It fetches all of the borrows that an account has made, collecting data such as the address of the token, its current price (denoted in ETH) and its liquidation threshold. Furthermore, it fetches similar data for the assets that an account has deposited into lending pools and therefore can be used as collateral (depending on the specific asset).

From the collected data, the overall lending position can be reconstructed allowing for the calculation of an account's health factor. The health factor is calculated according to the equation (2.1) detailed in the background chapter and loans with a health factor less than 1 are identified as liquidation opportunities.

The general architecture for the Liquidation Opportunity Oracle as well as its relation to the on-chain Community Pool Protocol is described in figure 3.1.

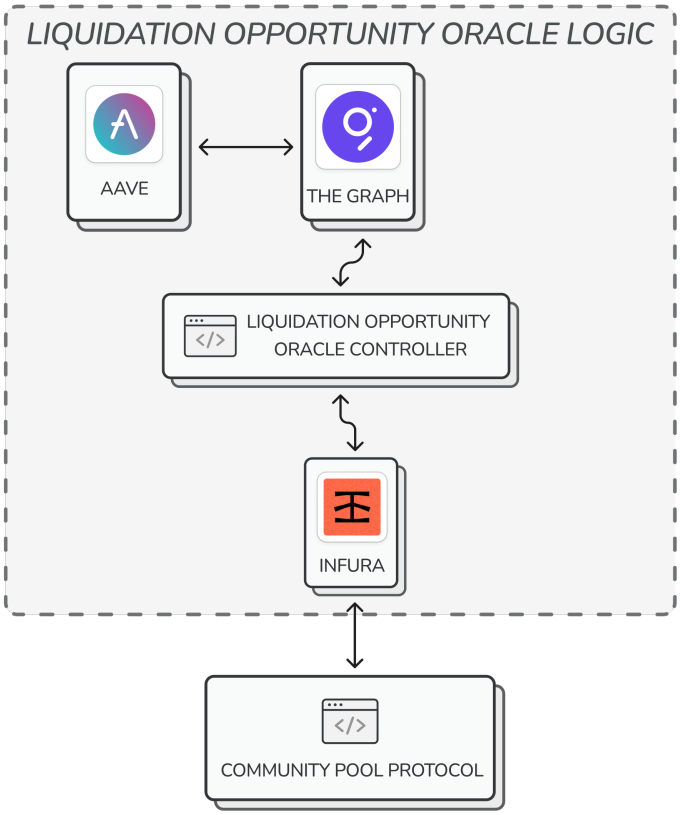

Figure 3.1: Liquidation Opportunity Oracle (LOO) Architecture.

When a liquidation opportunity is identified by the LOO, its profitability is analysed to ensure that it is worth liquidating. The LOO determines if the liquidation is worthwhile based on the following criteria:

- Is the amount to liquidate sufficient to make a meaningful profit after gas fees?
- Is the liquidation amount for the asset within the minimum and maximum liquidation amounts for that asset set by governance?
    - As detailed in the background chapter, price impact can affect the overall profitability of the process since trying to market sell too much of the discounted collateral asset - relative to the overall liquidity of the token in Uniswap pools - could lead to money being lost overall. This is why maximum liquidation amounts are set.
    - Too low of a liquidation amount would be unprofitable due to fees.

- Is the collateral asset that will be received from the liquidation on the governance determined allowlist? This is detailed more comprehensively in subsequent sections but it is based on the perceived risk associated with a particular asset.

After identifying a liquidation opportunity and confirming its suitability the LOO will utilise the Community Pool Protocol to call a function within the smart contract that executes the liquidation logic. This adds a level of centralisation to the design, however, it is necessary to construct the LOO in a centralised manner as a fully decentralised implementation would not currently be viable due to gas fees and the nature of how smart contract functions are called on the Ethereum blockchain. The impact of this centralised point of the system is mitigated by the protocol architecture (detailed in section 3.3.3) which restricts the power of the LOO, incentivises the desired behaviour and allows it to be governed by the decentralised stakeholders of the system.

3.3 Community Pool Protocol (CPP)

3.3.1 Executing the Loan Liquidation

When the LOO makes a call to the protocol to execute a loan liquidation, the parameters of the call are used to verify (on-chain) that it is indeed a valid liquidation opportunity. This is carried out by first retrieving the price of the debt asset via a decentralised price feed from Chainlink [10], fetching the Aave user account data to verify that the health factor is below 1 and finally, performing the necessary calculations to estimate the profitability of the liquidation i.e. verifying the amount of debt asset to liquidate as well as ensuring the amount is within the minimum and maximum liquidation amounts set by governance.

Following this, the appropriate amount of the debt asset is purchased on Uniswap [6] - a decentralised exchange - because Aave liquidations require that the liquidator already have a sufficient balance of the debt asset to be used to pay back the debt when executing the liquidation. Once the debt asset has been acquired, the liquidation can be executed by calling the relevant function within the Aave protocol's smart contract. The CPP smart contract directly calls Aave's "liquidationCall" function to carry out the liquidation on the protocol.

Aave loan liquidations allow the liquidator to receive either the underlying collateral asset or the aToken after a successful liquidation call. To avoid market risk, due to price movement, the underlying asset is chosen to be received so that it can be market sold as soon as possible on Uniswap. Once the liquidation process is complete and the discounted asset has been sold, the profit is calculated and distributed proportionately between the funders of the pool.

If for any reason, the liquidation does not work properly then the transaction is reverted and it is as though the whole execution of the loan liquidation stage had never happened in the first place. This functionality is due to Solidity's revert operation which reverts all changes to the state. Fig. 3 shows a flow chart representing the high-level overview of the smart contract with its revert functionality.

Moreover, it should be noted that the functionality of the protocol is restricted in that although the liquidation is triggered by a centralised oracle, the actions that it can perform are limited to only allow using the pool's funds to liquidate a loan - a loan that is verified on-chain to be worthwhile liquidating. This is of paramount importance for minimising the centralisation risk of the LOO and is bolstered by the capabilities of the governance system.

The general architecture of the Community Pool Protocol and its relation to the Liquidation Opportunity Oracle is shown in figure 3.2.

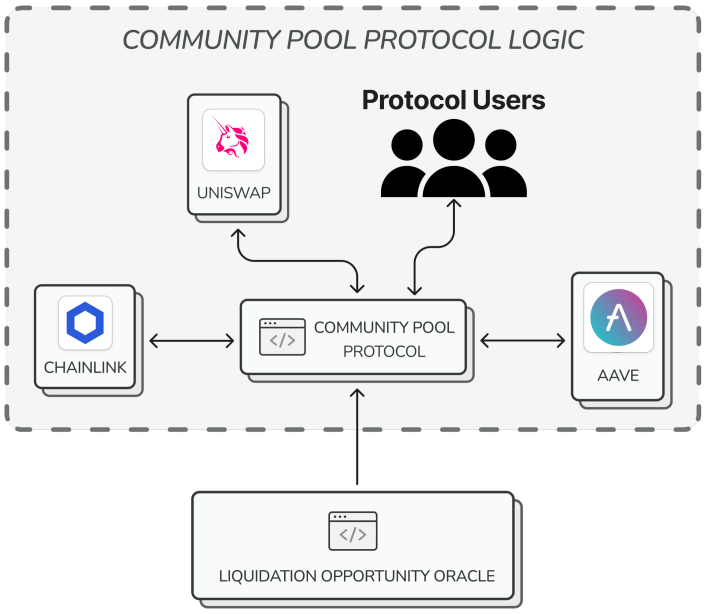

Figure 3.2: Community Pool Protocol (CPP) Architecture.

3.3.2 Smart Contract Architecture

The smart contract is the Ethereum program that handles the on-chain logic to verify and liquidate an unhealthy loan identified by the LOO. The CPP smart contract also facilitates the governance capabilities of the system as well as managing and storing the funds that the users of the protocol deposit into it.

The main function of the CPP smart contract is the "executeLiquidation" function which encapsulates the main purpose of the whole smart contract. The overall flow of the function is shown in figure 3.3.

The function "executeLiquidation" can **only** be called by the LOO whenever it identifies a liquidation opportunity. The following pieces of data are passed into the function from the LOO:

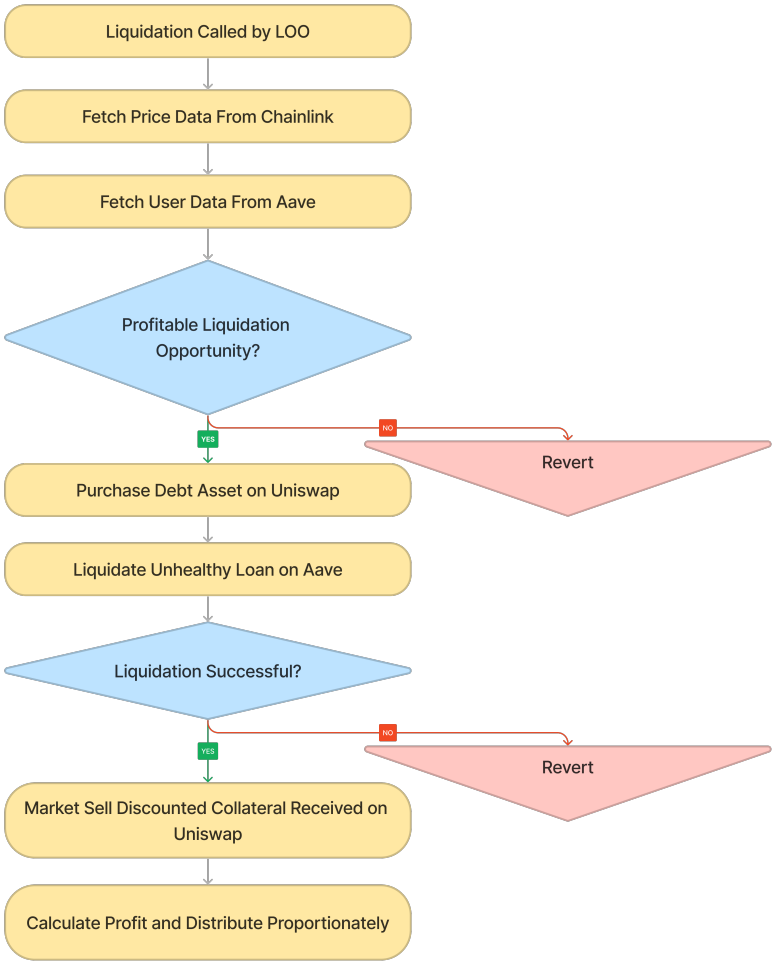

Figure 3.3: CPP Liquidation Call Overview Flow Chart.

- The address of the collateral asset.
- The address of the debt asset.
- The address of the user that has the related unhealthy lending position.
- An unsigned 256-bit integer representing the amount of debt that we would like to cover when liquidating the loan.

Straight away the "executeLiquidation" function ensures that the collateral asset that would be received on a successful liquidation is on the governance-determined allowlist. Once it performs this check, it then takes note of the current USDC balance of the contract; this is required for profit calculations at the end of the liquidation process.

Following this, the smart contract can now start to verify the liquidation opportunity on-chain by fetching the user's reserve data for the debt asset directly from Aave. Only a few of the data points related to the user's reserves are relevant for this verification:

- The total amount of liquidity borrowed at a stable rate represented in debt tokens [5] for the specific debt asset (SD^{asset}).
- The total amount of liquidity borrowed at a variable rate represented in debt tokens [5] for the specific debt asset (VD^{asset}).

The amount of debt that can be covered in the liquidation is calculated according to the following formula:

$$debtToCover = \left(SD^{asset} + VD^{asset}\right) \times LiquidationCloseFactorPercent \tag{3.1}$$

where the $LiquidationCloseFactorPercent$ is the maximum percentage of the debt that we can liquidate. This value is set by the protocol to currently be 0.5.

The reason this data is fetched on-chain as well as previously being fetched off-chain is to ensure that the passed-in data to the smart contract is still valid and has been passed with integrity by the Liquidation Opportunity Oracle. This heavily mitigates decentralisation risk preventing the Liquidation Opportunity Oracle from making false or malicious calls that do nothing else but waste users' funds in an attempt to disrupt the protocol. This form of on-chain verification reduces the power of the off-chain element of the system making it less susceptible to disruption by restricting functionality and verifying the integrity of the operation.

Once the contract has verified the lending position on-chain and the amount of debt that can be liquidated, it must acquire the required amount of the debt asset since the Aave protocol requires that the calling smart contract already has the asset when liquidating a loan. A couple of methods were considered when deciding how to implement such functionality.

One method of handling this would be to store reserves of the various assets that are predicted to be needed for future loan liquidations, however, this is a flawed approach for two reasons.

1. The risk associated with keeping reserves of volatile assets.
    - Consider a user that deposits 1000 US Dollars worth of some volatile cryptocurrency token that is to be used for liquidations. Users deposit into the pool with the expectation of earning a return based on the profit derived from the liquidation of loans. If volatile assets are allowed to be deposited then users may end up with less money overall even if there are successful liquidations due to the price movement of the volatile asset. This system design aims to allow users to earn a return solely from the liquidation process.
2. Reduced size of liquidations.
    - The profit earned from a loan liquidation is related to the overall size of the liquidation. If the funds controlled by the system are partitioned into various cryptocurrency tokens then this reduces the size of liquidation that can be performed at once and therefore the potential profits.

Another method - and the chosen solution - is to purchase the debt asset when it is required, though, this means that the CPP smart contract must have access to funds that can be used to purchase the debt asset on-chain. This functionality is facilitated by storing the reserves of the CPP smart contract in a stablecoin. These stablecoin reserves that have been deposited into the CPP smart contract by the pool participants can then be used to purchase the debt asset required for a liquidation providing benefits such as:

- Less exposure to volatility risk. Volatile assets are more likely to be viable since the system would only use/hold them for short periods of time.
- Stablecoins tend to be highly liquid in automated market maker pools meaning that it is relatively easy to purchase a variety of assets that may be debt assets for liquidations using them.
- Increased size of liquidations. The amount of debt asset acquired is only limited by the CPP's funds and the availability of the asset to be purchased rather than relying on the CPP smart contract to keep the asset itself.

This leads to the question, "which stablecoin should be used in the system?".

According to Moin et al., "Stablecoins are a class of cryptoassets created to provide the stability money needs to function. As the name implies, they are designed to be price stable with respect to some reference point, such as USD." [23]. Whilst there are many types of stablecoin [29], two of the most prominent cryptocurrency stablecoins are US Dollar Tether (USDT) and US Dollar Coin (USDC) - which are fiat-collateralised stablecoins, both pegged to the US Dollar - with respective market caps of $70.5 billion and $42 billion according to CoinMarketCap on the 21st of February 2023 [12].

The scheme detailed in this paper opts to use USDC as its stablecoin due to controversies surrounding USDT's collateral [25] with Stuart Hoegner who is general counsel for Tether stating in the Supreme Court of New York that USDT was only 74% collateralised [19]. The under-collateralisation of USDT poses a de-pegging threat (the risk of the value not being linked to 1 US dollar) to the stablecoin which could have massive knock-on effects for the proposed system had it chosen to use it.

Storing the pool's funds in USDC means that the system has a stable store of value that generally can be converted to another token with minimal price impact due to the high liquidity of stablecoin-paired pools on exchanges like Uniswap.

Now that the system has a stable store of funds, the "executeLiquidation" function can use these stable funds to acquire the debt asset by purchasing it through a decentralised exchange.

Xu et al. present a comparison table (table 3.1) in their paper regarding decentralised exchanges (DEXs) with automated market makers (AMMs) [31]. This table details some important aspects to consider when determining which decentralised exchange to use for the system.

Uniswap is the DEX of choice for this system for a few reasons, such as, it being the largest DEX by market share, it having the highest number of governance token holders and it having the most trading volume. These attributes mean that Uniswap is a

Protocol	Value locked ($bn)	Trade volume ($bn)	Market (%)	Governance Token	Governance token holders	Fully diluted value ($bn)
Uniswap	6.15	11.4	66.7	UNI	269,923	21.1
Sushiswap	3.92	2.9	14.2	SUSHI	71,007	2.4
Curve	11.64	1.8	6.4	CRV	44,654	4.0
Bancor	1.37	0.4	2.5	BNT	38,124	0.8
Balancer	1.74	0.5	2.2	BAL	37,613	1.0
DODO	0.07	0.4	2.1	DODO	11,330	1.2

Table 3.1: "Comparison table of discussed DEX: value locked, trade volume of the past 7 days, the market share by the last 30 days volume, the governance token, the number of governance token holders and the fully diluted value, as on 21/09/2021. Data retrieved from DeFi Pulse and Dune Analytics." [31]

well-documented protocol that is suitable for the CPP smart contract to interact with as well as having highly liquid pools meaning that large volumes of assets can be traded with minimal price impact. In addition to this, the high number of governance token holders suggests that it is a highly decentralised protocol which helps to preserve the degree of decentralisation of the proposed system.

Uniswap has two functions within the smart contracts of its protocol that are relevant to the proposed system:

1. "exactOutputSwap": This allows a token to be purchased, specifying the exact number of tokens that must be received.
2. "exactInputSwap": This allows a token to be purchased, specifying the exact number of tokens that will be used to purchase the other token.

Due to the AMM mechanism - detailed in the background chapter of this paper (2.2) - the price that is paid for a single unit of a token is not the same throughout the purchase of another token on Uniswap. This is due to things such as price impact and slippage. Often, systems need to spend a certain amount of tokens *or* receive a certain amount of tokens; Uniswap provides these options via the two aforementioned functions.

When acquiring the debt asset for the Aave lending position liquidation, the system requires an exact amount of the debt asset; the amount of debt asset to cover. For this reason, "exactOutputSwap" is used to receive the exact amount of output tokens.

This function has a parameter, "amountInMaximum", that determines the maximum amount that is willing to be paid for the given exact amount of output tokens. This is to avoid massively overpaying for a trade. If no limit was given then the trade is susceptible to being front-run (described in the evaluation chapter (section 4.4) or a poor price being paid due to rapid price fluctuations in the market. It is much safer to set the "amountInMaximum" value based on recent price data to avoid these risks and that is why the proposed system sets this value using a price oracle.

To set the "amountInMaximum", recent price data is required from a decentralised price oracle. For this, the CPP smart contract uses Chainlink and calls their price feed oracle requesting real-time price data for the debt asset in USDC. Chainlink provides various data feeds for the price of tokens. The address for the Chainlink oracle is stored in a struct - named AssetParameters - defined in the CPP smart contract that contains information such as the minimum and maximum liquidation amounts and whether the asset is allowed or not.

Once this price value has been received, the CPP smart contract multiples this price by the amount of debt asset required and adds a 2% leeway value to allow for small market fluctuations increasing the probability that the token swap is successful.

the "executeLiquidation" function can now call "exactOutputSwap" on the Uniswap protocol to receive the debt asset. After this, the contract can begin to liquidate the unhealthy lending position. Due to how ERC-20 tokens work on the Ethereum blockchain, the CPP smart contract must approve the transfer of its debt asset token. There are a couple of methods that can be used to do this, the proposed design makes use of the "safeApprove" function. This function is from the SafeERC20 library which provides wrappers around the ERC20 token functions.

SafeERC20 was used here (and also in other parts of the CPP smart contract code) since its functions increase smart contract security by performing checks that ensure the tokens were "approved" and "transferred" properly. As well as this, using SafeERC20 saves development time compared with manually writing these checks ourselves in the smart contract.

After the CPP smart contract has approved the Aave protocol's lending pool address to take the debt asset tokens from it, the liquidation is started by calling the "liquidationCall" function of the Aave lending pool smart contract. The relevant data is passed into the "liquidationCall" function, specifically:

- The token address of the debt asset.
- The token address of the collateral asset.
- The address of the Aave user with the unhealthy lending position.
- The amount of debt asset to liquidate.
- A boolean value representing whether to receive the AToken balance for the collateral on successful liquidation.

The CPP smart contract sets the last argument to "false" since one of the design decisions is to receive the actual collateral asset so that it can be market sold as soon as possible. This is desirable as it avoids market risk from fluctuations in the price of the collateral asset by selling it at the market price straight away.

At this stage, provided the information passed to the "liquidationCall" function call was correct - which it should be since it has been verified - the liquidation will be successful and the CPP smart contract will receive the discounted collateral asset. If, for any reason, the liquidation fails then the "executeLiquidation" will take advantage of the revert functionality in Solidity and revert the whole transaction so that none of the state changes executed in the function will be written to the blockchain. This prevents the system from being left with the debt asset after a failed liquidation, instead, it is as though the "executeLiquidation" function was never called in the first place (minus the gas fee paid for partial execution).

It is now time for the collateral asset to be sold. For this, the other Uniswap function mentioned previously, "exactInputSwap" is used. This allows the exact amount of the collateral asset that has been acquired via the liquidation to be sold. The CPP

smart contract checks its balance of the collateral asset by calling the "balanceOf" function of the ERC20 token. Since the CPP smart contract stores its funds in the stablecoin, USDC, the collateral asset will be sold for USDC. Similarly to purchasing the debt asset on Uniswap, the "exactInputSwap" function takes a parameter called, "amountOutMinimum", which determines the minimum amount of the token that would be accepted for the given exact amount of input tokens. Again, this value must be set based on recent price data fetched from Chainlink. The process is the same as described for the purchase of the debt asset with a 2% leeway value.

Finally, the "executeLiquidation" function checks its USDC balance and compares it with the value recorded at the start of the function call, allowing for the calculation of the overall profit of the process. This profit must be distributed between the pool participants of the system proportional to their stake i.e. their deposit. This could be performed by looping through all of the pool participants, calculating their share and allocating it to them but this would be highly inefficient as explained in section 4.3.2. Instead, a uint256 variable named "poolMultiplier" is used to store the relative growth of the CPP smart contract's pooled funds due to liquidation profit. This is the method chosen for the design of the CPP smart contract since it is significantly more gas efficient and thus more economically viable.

The "poolMultiplier" value is effectively set at 1 at the inception of the smart contract. As the system grows and makes a profit, this value will grow i.e. if the pool had made 20% profit from a liquidation call, the new "poolMultiplier" value would be 1.2. Solidity doesn't fully support floating point numbers so it must be stored as an integer value with a set number of decimal places. For description purposes, the example shown here just deals with the floating point representation, though, the CPP smart contract implements the same functionality using fixed decimal calculations.

The "poolMultiplier" variable is important for understanding how the profit and funds of the system are tracked as well as for explaining how the system handles deposits and withdrawals.

To demonstrate how the "poolMultiplier" works, consider a scenario such that the CPP smart contract pool has a total of 9000 USDC in it from several depositors and previous operation of the system has caused the "poolMultiplier" to be 1.1. Bob deposits 1000 USDC to the CPP smart contract meaning they have a 10% share of the pool. The "deposit" function of the CPP smart contract accepts the transfer of funds to it and adjusts internal variables to store the user's participation in the system. Bob's share of the pool will be stored relative to the "poolMultiplier" and calculated as follows:

$$\begin{aligned} poolShare_{user} &= \frac{depositAmount_{user}}{poolMultiplier} \\ &= \frac{1000}{1.1} \\ &= 909.09 \end{aligned} \tag{3.2}$$

The "withdraw" function of the CPP smart contract only lets a user withdraw their share of the pool and calculates the withdrawable amount similarly to the "deposit" function

again making use of the "poolMultiplier". If the user were to withdraw their funds, the amount they could withdraw is calculated as follows:

$$\begin{aligned} withdrawableAmount_{user} &= poolShare_{user} \cdot poolMultiplier \\ &= 909.09 \times 1.1 \\ &\approx 1000 \end{aligned} \tag{3.3}$$

Consider that the user has deposited and not withdrawn, the system then liquidates an unhealthy loan earning 500 USDC in profit. Bob's earnings would be 50 USDC i.e. 10% of the profit so his new pool share is 1050 USDC. To update the share so that it reflects this earning, the "poolMultiplier" must be updated according to the following equation:

$$\begin{aligned} newPoolMultiplier &= \frac{TotalPoolValueAfter \cdot currentPoolMultiplier}{TotalPoolValueBefore} \\ &= \frac{10,500 \times 1.1}{10,000} \\ &= 1.155 \end{aligned} \tag{3.4}$$

If Bob now wanted to withdraw his funds and receive his earnings, the withdrawable amount would be calculated using equation 3.3 with the new poolMultiplier value ($909.09 \times 1.155 \approx 1050$). This means that the share that the user owns of the pool can be dynamically calculated at the time it is needed and lots of expensive computational power is saved. This is why this method was chosen over the expensive looping method.

Figure 3.4 details a UML sequence diagram for the typical flow of the Liquidation Opportunity Oracle and the Community Pool Protocol. The LOO discovers an unhealthy lending position by querying The Graph, and passes the information to the CPP which verifies the data on-chain, acquires the debt asset on Uniswap, liquidates the loan on the Aave protocol, sells the collateral asset on Uniswap and finally updates the CPP smart contracts internal data.

3.3.3 Governance Abilities

To allow stakeholders to maintain the integrity of the system and ensure it acts in their collective best interest, pool participants are given voting rights weighted on the proportion of their share of the pool. Governance abilities include:

- Determining the Liquidation Opportunity Oracle (LOO) caller address. The stakeholders have the power to change the address that is allowed to execute liquidations on behalf of the community. Reasons for doing so could be:
    - The current LOO is making incorrect or malicious calls.
    - The community have developed a better implementation of the LOO and wants to switch it out for the new one (the LOO is modular in this sense).

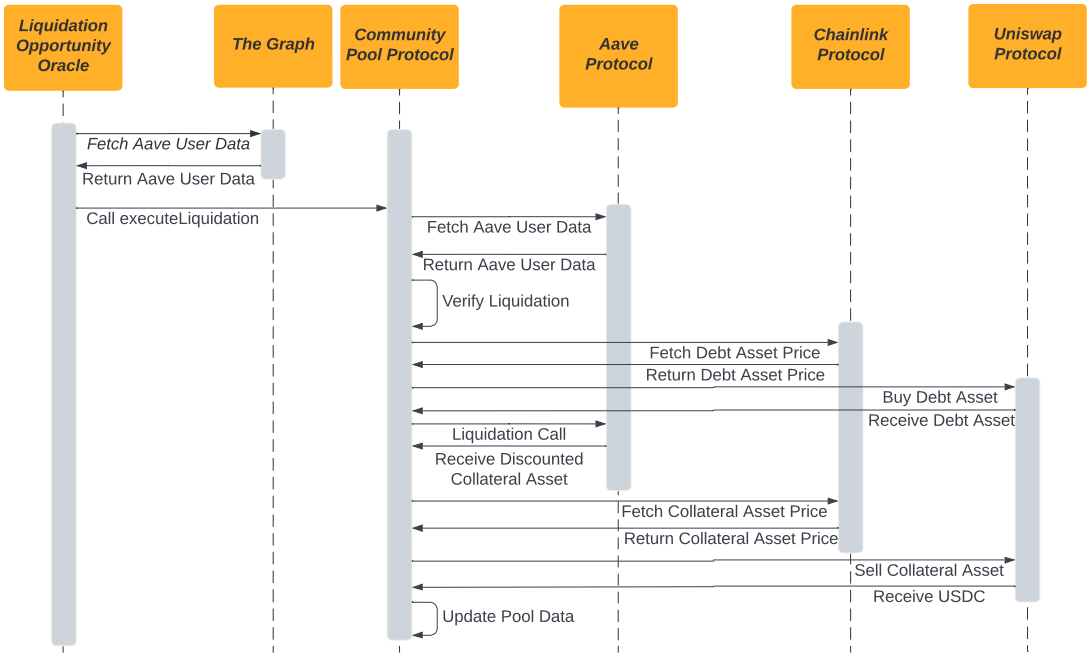

Figure 3.4: UML sequence diagram for the liquidation process of the proposed system.

- Altering the parameters used for calculating a profitable liquidation. For example, governance may decide that loans below a certain value are not worthwhile liquidating.

Governance is carried out via a voting system that delegates voting power proportionately to a stakeholder's share of the pool. This voting power is calculated using the share of the pool which comes from the "poolMultiplier" variable previously discussed.

Governance votes can be proposed by stakeholders, however, proposing a change requires a monetary deposit, encouraging only necessary proposals. The fee amount set in the CPP smart contract is 0.01 ether (approximately 18$). This is a sufficient amount to discourage bad actors from attempting to DOS attack the system by needlessly calling governance proposals since it would cost them every time to do so but not so much to discourage genuine participants from making a needed governance proposal. Furthermore, the design of the CPP smart contract implements functionality to allow proposers to reclaim this fee if their proposal is successful. Again, this promotes honest usage of the proposal system.

To implement the governance system, the CPP smart contract makes use of an enum to store the different types of governance proposals. The values of the "ProposalType" enum are as follows:

- ALLOWLIST_UPDATE - This represents the update of the boolean value repre-

senting whether a specific asset is allowed in the system.

- UPDATE_LOO - This represents the update of the LOO address.
- UPDATE_MIN_LIQUIDATION - This represents the update of the minimum liquidation amount associated with a specific asset.
- UPDATE_MAX_LIQUIDATION - This represents the update of the minimum liquidation amount associated with a specific asset.

In addition to this, the design uses an array of struct to store the proposals made for the system. The struct stored in the array is called "Proposal" and has the following fields:

- proposedChange - One of the ProposalType enums.
- votingPeriodEndBlock - The block number at which voting on the proposal will end.
- forVotes - The number of votes for the proposed change.
- voted - A mapping that is used to store which addresses have voted for the proposal to avoid double voting.
- accepted - A boolean representing whether or not the proposal has been accepted.
- implemented - A boolean representing whether or not the proposal has been accepted.
- feeCollected - A boolean representing whether or not a successful proposal's fee has been reclaimed by the proposer.
- proposer - The address of the proposer.
- relatedAddress - The address related to the proposed change. Used for certain proposal types.
- relatedValue - The value related to the proposed change.
- minForVotesRequired - The minimum number of for votes required for the proposal to be passed by governance.

The Proposal struct and the ProposalType enum are shown in figure 3.5.

When a user wishes to propose a change to the system, they must call the "createProposal" function, passing it the values related to the change as well as paying the 0.01 ether fee. This creates a Proposal struct with the relevant data, setting the minimum number of for votes to 50% of the voting power of the system. Essentially, for a given proposal to be passed by governance, more than 50% of the CPP smart contracts stakeholders (measured by the size of their stake) have to vote for the change. This means that whatever the majority of the system's stakeholders want will be passed. When calling the "createProposal" function the return value is the ID of the proposal, which other users can use when voting.

Once a proposal has been created the participants of the system can vote on the proposal by calling the "castForVote" function and passing the "proposalId". The function

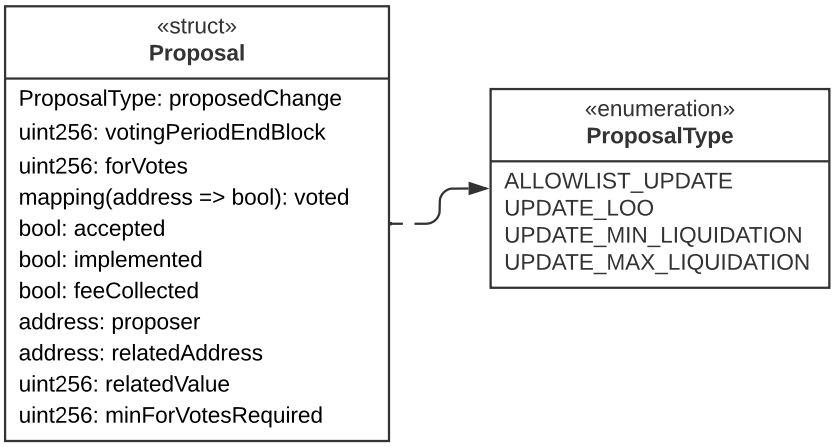

Figure 3.5: UML class diagram for the Proposal struct and the ProposalType enum.

ensures that the voter has not already voted and that the voting period is still active.

The block that the voting period will end at is calculated by adding the number of blocks that a voting period should last to the block number when the proposal is created. The CPP design chooses to set the "votingPeriodEndBlock" value by adding 40,320 to the current block number since 40,320 blocks at an average block time of 15 seconds on the Ethereum mainnet is roughly 1 week. This value is chosen since it is not too short to make it difficult to vote in time but not too long that the voting period runs on needlessly.

When the voting period has ended and if the number of "for" votes for a proposal was sufficient, the "implementProposal" function can be called by anyone. The decision to allow the function to be called by anyone increases the chance that it will be called. This is useful since if a proposal passes, it has been deemed to be what the stakeholders of the system want. Restricting the calling of this function to, for example, only be callable by the proposer means that they can change their mind after the fact or for any other reason not call the function and the change wouldn't be implemented.

The "implementProposal" function figures out the type of proposed change using the related ProposalType enum and updates the system accordingly.

Finally, the proposer of a successful vote can recoup their proposal creation fee by calling the "withdrawSuccessfulProposalFee" function which sends them back their 0.01 ether.

The governance system of the CPP smart contract allows for the decentralised management of the protocol and bolsters the level of decentralisation of the entire proposed scheme.

Some of the anticipated reasons that the community of the system would propose a change, particularly, related to the parameters for an asset that the system deals with would be:

- The volatility of the asset.

    - Volatile assets are riskier to deal with, so the governance of the system may decide to not allow these assets to be used in the system to protect its stakeholders.

- The amount of liquidity and accessibility of the token on Uniswap.
    - If an asset is not very liquid on a decentralised exchange on Uniswap, a trade executed to purchase or sell it could be heavily vulnerable to substantial losses due to price impact and so the governance of the system may opt to protect from this by either banning the asset or carefully selecting the amount of the asset that can be handled in one liquidation.
- The overall risk of the token to catastrophic events such as decentralisation risk which could cause the token to completely collapse.
- General perceived risk due to technical implementation or the team that manages the token design.

A UML class diagram is presented in figure 3.6 that shows more detail regarding the implementation of the Community Pool Protocol.

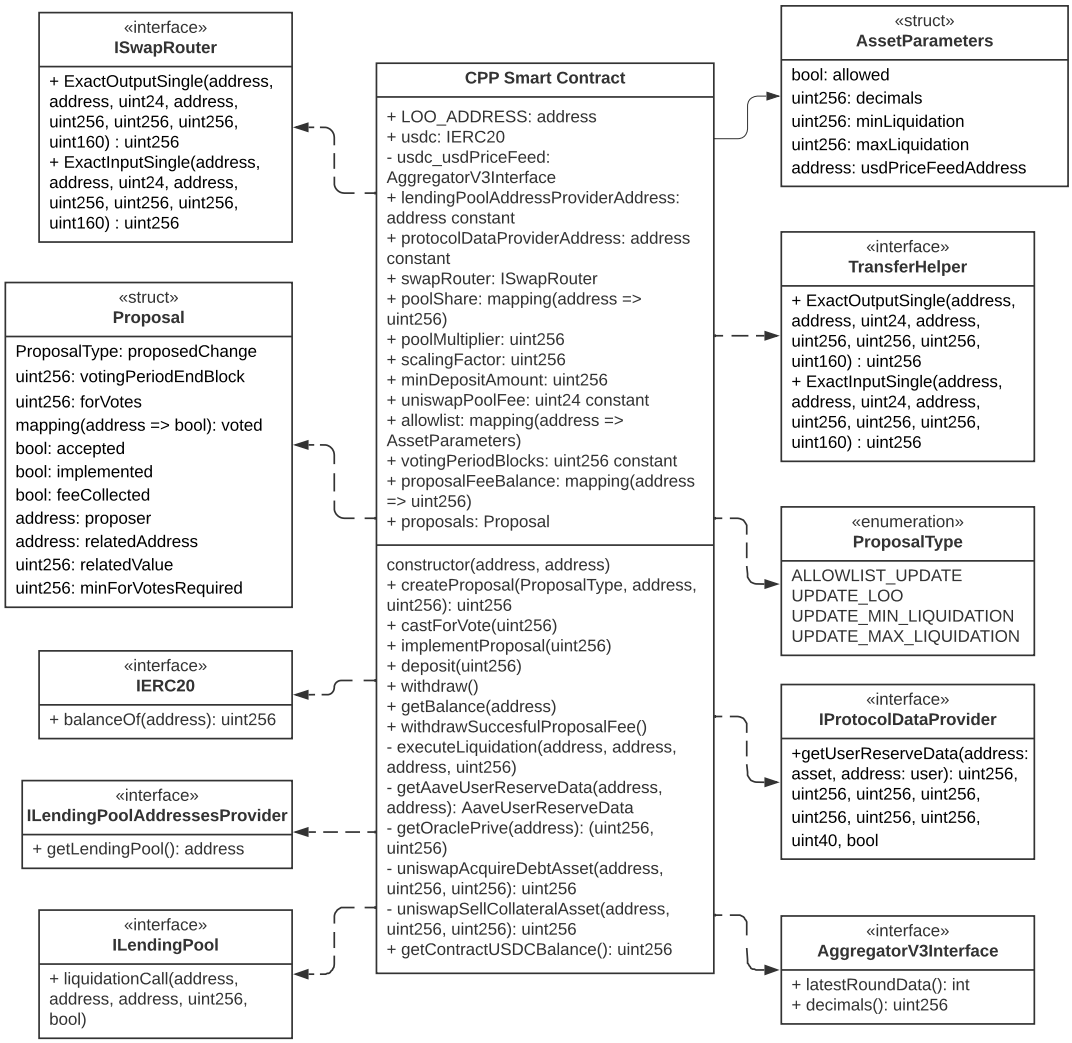

Figure 3.6: UML class diagram for CPP smart contract.

Chapter 4

Experiments and Evaluation

In this chapter, the experimental setup is described and the results are discussed and evaluated. Testing complex interactions between various Ethereum protocols is a challenging endeavour due to the documentation for the protocols tending to only cover extremely basic scenarios. A significant amount of time was dedicated to reading through the smart contracts of these protocols to design an experimental environment that facilitated the testing of the proposed system. The experimental setup consists of creating a local fork of the Ethereum mainnet and simulating the required Ethereum blockchain conditions required for a loan liquidation to take place. The latency of the "executeLiquidation" function is explored and the gas efficiency of the smart contract is analysed and evaluated. Furthermore, the security- and decentralisation-design of the system is evaluated.

4.1 Experimental Setup

4.1.1 Experimental Environment

To test the performance of the smart contract a local environment was set up. The Ethereum development environment, Hardhat [3], was used to create an instance of the Hardhat Network that forks the Ethereum mainnet allowing the experiments to operate in an environment that simulates having the same state as the mainnet but works as a local development network. This is required when working with smart contracts that need to interact with deployed protocols as it facilitates the testing of complex interactions in a local environment.

To create a mainnet fork, a network provider is required, thus an Infura [1] node was used. Once the local mainnet fork was set up a Hardhat project was created with the Community Pool Protocol smart contract code.

A test suite was set up using ethers.js [2] to interact with the smart contract and mainnet fork, and Mocha [4] was used as the test runner.

The local mainnet fork was run on a 2021 MacBook Pro with an Apple M1 Pro chip and 16 GB of memory.

4.1.2 Simulating a Price Drop

To test the loan liquidation functionality of the CPP smart contract, the relevant Ethereum network conditions on various protocols had to be simulated. As detailed previously, a loan position is available to be liquidated when the health factor of a user drops below 1. The Aave protocol calculates this health factor according to equation 2.1 which is dependent on the value of the debt and collateral assets that a user has.

Ethers.js was used in the test suite to impersonate a signer (note: this is a testing method that can only be performed in local blockchains) for the Ethereum address of the Aave protocol, namely the PoolDataProvider. This address manages the data that the Aave protocol uses to calculate attributes such as the health factor. The PoolDataProvider was impersonated to sign a change to the oracle address that is used by the Aave protocol to fetch price data about WETH - an ERC-20 token.

The oracle address for WETH was changed on the Aave protocol to an address that returned a significantly lower value than the actual price of the WETH token, thus, making the Aave protocol think that the price of the WETH token had dropped massively.

This impacts the health factor of users that are using WETH as a debt or a collateral asset on Aave. Users that are using WETH as a collateral asset will have their health factors impacted negatively and in some cases reduced below 1 so that their position is now unhealthy and can be liquidated.

As of writing, the real price of WETH - which is linked 1-to-1 with ETH - is around 1,700$. The experimental setup changes the oracle for WETH to the oracle for MKR - another ERC-20 token - which is currently priced at around 600$ (a roughly 65% drop).

4.1.3 Testing the Liquidation Call

To test the liquidation of a loan position via the CPP smart contract some basic prerequisite steps were taken.

The test environment first deploys the CPP contract to the local mainnet fork calling the constructor method of the contract and setting the address of the LOO. Once this is done, the contract can be accessed in the local environment as though it was actually on-chain.

An Ethereum address to be used as a test pool participant address was transferred 1,000,000 USDC from a USDC whale account (an account with a large number of tokens) on the mainnet fork. The whale account was identified on Etherscan [15] by looking at the top holders (by volume) of the USDC token. This address was then impersonated using ethers.js and used to send the USDC to the test pool participant address. This test address then deposited the 1,000,000 USDC into the CPP smart contract via the deposit function.

The Liquidation Opportunity Oracle was used to identify a user that had used WETH as collateral and therefore their health factor had dropped significantly once the WETH oracle address was altered.

The tests then call the “executeLiquidation” passing in the appropriate data to liquidate the unhealthy Aave loan position of the user. Some sanity checks are made within the tests to ensure that the loan was liquidated correctly and a profit was made.

Carrying out this process allowed for the exploration of the gas efficiency of the smart contract as well as the latency of the “executeLiquidation” function.

4.2 Latency of the executeLiquidation Function

Speed is important when it comes to the liquidation of loan positions since it could be a sharp drop in price that triggers the unhealthy position but this price drop will not necessarily sustain. An efficient liquidation system must execute the liquidation swiftly to maximise its earning potential.

For this reason, the latency of the “executeLiquidation” function is of interest. To measure the latency, the testing environment measured the time taken for a liquidation execution function to be called and executed in the mainnet fork.

The latency value for the “executeLiquidation” function of the CPP smart contract was compared to the latency value for manually calling the core Aave “liquidationCall” function directly with the input data. Both methods were run 10 times each on fresh local Ethereum mainnet forks at a specific block number to ensure the results were comparable and not impacted by network congestion. The block number used was 16928579.

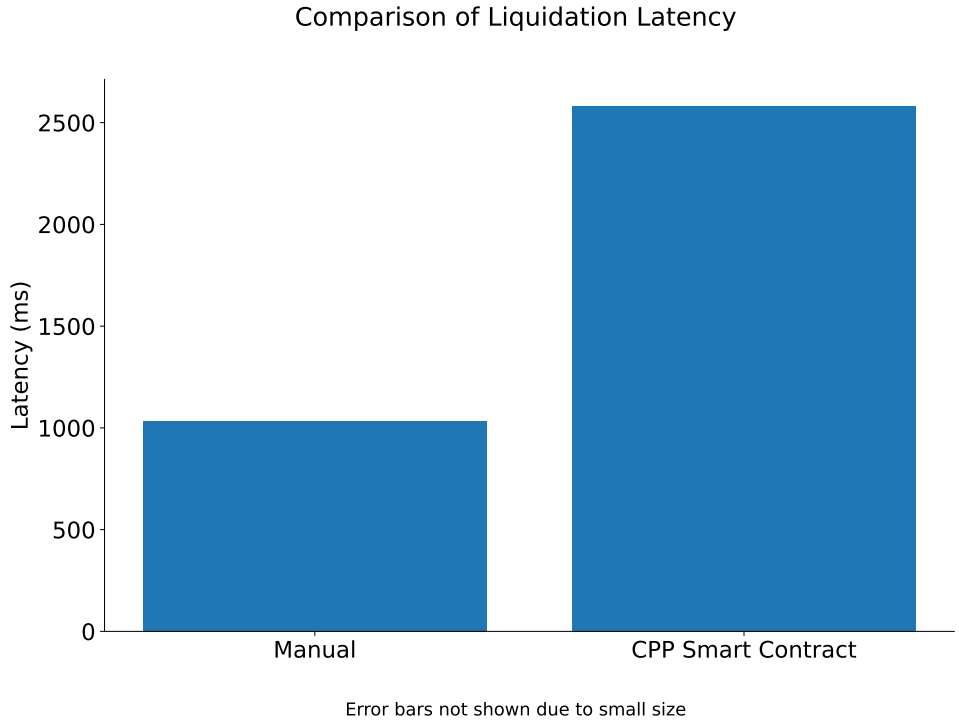

Figure 4.1: Comparison of latency values for manual liquidation and liquidation via the CPP smart contract.

It is clear that the CPP smart contract's "executeLiquidation" function has a higher latency than the latency of manually calling Aave's "liquidationCall" function. This is expected since the "executeLiquidation" function calls "liquidationCall" within it. "liquidationCall" had an average latency of 1035.27 milliseconds whereas "executeLiquidation" had an average latency of more than twice that at 2583.13 milliseconds. Whilst "executeLiquidation" does significantly add to the latency value, the trade-off must be considered in the context. "executeLiquidation" performs an on-chain verification of the liquidation parameters ensuring that the liquidation is viable, as well as this, it must also safely acquire the debt asset and safely sell the discounted collateral in this time. This makes for a much safer liquidation by performing sanity checks as well as market selling the collateral asset to avoid market risk. The added latency overhead is a small price to pay for the benefits received and required by the proposed system.

4.3 Gas Analysis

Gas efficiency is the most important performance aspect of a smart contract. Gas is a measure of the computational resources required for the execution of a specific operation or transaction in a smart contract [32]; it is the unit of measurement for the amount of work that must be done by the Ethereum network to execute the given code.

Furthermore, the gas efficiency of a smart contract is also important since gas has economical value. The more gas it costs to execute a function, the more money it costs to run that function on the Ethereum network. This is a mechanism of the Ethereum blockchain to incentivise the efficient usage of the shared and rivalrous computational power.

In the context of the proposed system, gas efficiency is particularly important since the amount of computational power used to execute a liquidation will impact the overall profit earned by the system. Moreover, if the success rate of the liquidation aspect is not great enough relative to the gas cost - even for a failed execution - then the overall system would be unprofitable.

Many methods are employed to strengthen the gas efficiency of the smart contract design of this system.

4.3.1 Early Exit Require Statements

The smart contract makes use of early exit require statements. Require statements in solidity ensure that a certain condition is met and if not, will revert the transaction and stop the execution of the code.

Require statements are used throughout the smart contract code to validate inputs and ensure conditions for liquidations are met. These require statements are gas efficient since they are placed at the beginning of the functions (where applicable) so that they can be evaluated as early as possible and reverted (if need be). Reverting early means that the rest of the code after the require statement does not need to be executed as it would be pointless anyway since the transaction would ultimately fail but use up gas. This can end up saving a large amount of gas, particularly in the "executeLiquidation"

function which makes use of a few early require statements to ensure governance parameters are met and the liquidation is valid. One example of this is shown below in which a require statement is used at the very beginning of the "executeLiquidation" function to ensure that the collateral asset is marked as allowed by governance. This stops the function from carrying out unnecessary computations since it would eventually not work due to the asset not being allowed.

```
function executeLiquidation(
        address collateralAsset,
        address debtAsset,
        address user,
        uint256 debtToCover
    )
        external onlyLOO
    {
        require(allowlist[collateralAsset].allowed == true, "
            Collateral asset is not allowed.");

        // The rest of the executeLiquidation function.
        // ...

    }
```

The choice to use require rather than assert for such functionality is also gas efficient since require statements refund the gas on a failed require compared to assert which when it fails uses up the remaining gas.

4.3.2 Data Structure and Variable Usage

Another way that the smart contract was designed with gas efficiency in mind was through the careful usage of appropriate data structures.

One of the most computationally expensive processes in Solidity is using loops to iterate through data. For example, when storing the allowlist for the assets that the CPP smart contract deals with, a mapping from an address to a struct - called AssetParameters - has been used to map the token address of an asset to its relevant data such as the minimum and maximum liquidation amounts.

```
struct AssetParameters {
        bool allowed;
        uint256 decimals;
        uint256 minLiquidation;
        uint256 maxLiquidation;
        address usdPriceFeedAddress;
    }

mapping(address => AssetParameters) public allowlist;
```

This is more gas efficient than using an array to store a list of AssetParameter structs since to use this structure the code would need to iterate through the list attempting to match the given token address to the relevant struct in the array. The length of this array could be very large meaning it would cost a significant amount of gas to find tokens that are far into the array.

Using a mapping allows the AssetParameters to be identified by directly passing in the address to the mapping variable, as shown below. For example, if the smart contract needs to find the "usdPriceFeedAddress" for the USDC token - USDC token address: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 - it would access it as shown below.

```
// Accessing the usdPriceFeedAddress for the USDC token.
address UsdcUsdFeed = allowlist[0
    xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48].usdPriceFeedAddress;
```

This technique is used in various other parts of the CPP smart contract to reduce the computational complexity and therefore the gas used.

Moreover, another technique that the CPP smart contract employs to improve the overall gas efficiency is by using the "poolMultiplier" variable to track the economical performance of the system. The earnings that the system makes via loan liquidations could be tracked by just looking at the overall assets that it owns, though, this value is not representative or useful when calculating what an individual user's share of the pool is since they can deposit and withdraw at different times of the system's lifespan. When the CPP smart contract liquidates an unhealthy lending position it will make a profit and this profit must be shared proportionately between the pool participants based on their stake.

One method the smart contract could use to handle this functionality would be to calculate the share of the profit earned via a liquidation event for each individual user relative to their deposit amount as soon as the profit is calculated in the "executeLiquidation" function. For a system with many depositors, this would be highly computationally inefficient and reduce the overall profitability of the system due to gas fees. It would have to iterate through each depositor, calculate their share and update their balance.

The method chosen for the system is far more gas efficient as it instead uses the "poolMultiplier" variable to track the fluctuations in the pool value so that a user can dynamically calculate their share of the pool's funds. Instead of having to iterate and distribute earnings, the "poolMultiplier" can just be updated at the end of the "executeLiquidation" function so that the share of the funds owned by a user can be calculated using this value.

The calculation used to determine the balance that a specific user has in the system is shown in equation 3.2. The end of the "executeLiquidation" function updates the "poolMultiplier" as follows.

```
// Update pool multiplier
// getContractUSDCBalance() is effectively the poolBalanceAfter
poolMultiplier = (getContractUSDCBalance() * scalingFactor) /
    poolBalanceBefore;
```

4.3.3 Further Gas Efficiency Techniques

Some further gas efficiency techniques utilised in the design of the smart contract include:

- Use of constants.
    - Constants have been used for address values that will never change such as “lendingPoolAddressProviderAddress” and“protocolDataProviderAddress”. Doing this tells the Ethereum Virtual Machine (EVM) that these values will never change and allows it to perform gas optimisations.
- Use of memory variables.
    - Declaring variables as “memory” instead of “storage” means that the EVM knows to just store the data in the transaction memory rather than executing read and writes to storage which costs a lot more gas.
- Reusing variables.
    - Twice in the “executeLiquidation” function oracle price data must be fetched from Chainlink. From the oracle data, two values are received each time: the “price” and the number of “decimals” for the price value. Instead of using separate variables to store these values, they can just be reused since by the time the second Oracle call is made the first values are no longer required. This is more gas efficient than using multiple variables.
- Usage of SafeERC20.
    - The CPP smart contract uses functions from SafeERC20 rather than the original ERC20 functions. For example, the function “safeApprove”is a gas-efficient method for approving an ERC20 token transfer since it checks if the token already has an allowance and avoids overwriting if it does.

4.3.4 Gas Values for CPP Smart Contract Functions

The testing environment was used to execute and calculate the gas used for the main public functions of the CPP smart contract.

Figure 4.2 shows the difference between the gas used for the functions. Clearly, “executeLiquidation” uses the most gas but this is to be expected since it is undoubtedly the most computationally costly function of the smart contract. The second highest gas used by a function was the “createProposal” function which has to set and store a lot of data in storage and so uses a fair amount of gas. The other four functions use significantly less gas since they are less complex.

Gas fairness is an important aspect of a smart contract that will be used by a community of users. The CPP smart contract was designed with this in mind, making sure to spread the gas cost of the operation of the system between the users as fairly as possible. Whilst the “executeLiquidation” function uses significantly more gas, this gas cost is spread between the many users of the CPP smart contract so the burden does not lie on one individual party. Furthermore, there is a small amount of gas unfairness for the “createProposal” function but this is unavoidable. The “createProposal” function is used to allow the system’s users to propose a change to the system. This is required to ensure the system acts with integrity and operates efficiently, so it is a small price to pay to be able to change aspects of the system. Further work could improve the gas fairness of

the design by adding functionality so that the creator of a successful proposal can claim a gas rebate and gain back some of their funds spreading the cost among the rest of the pool's participants.

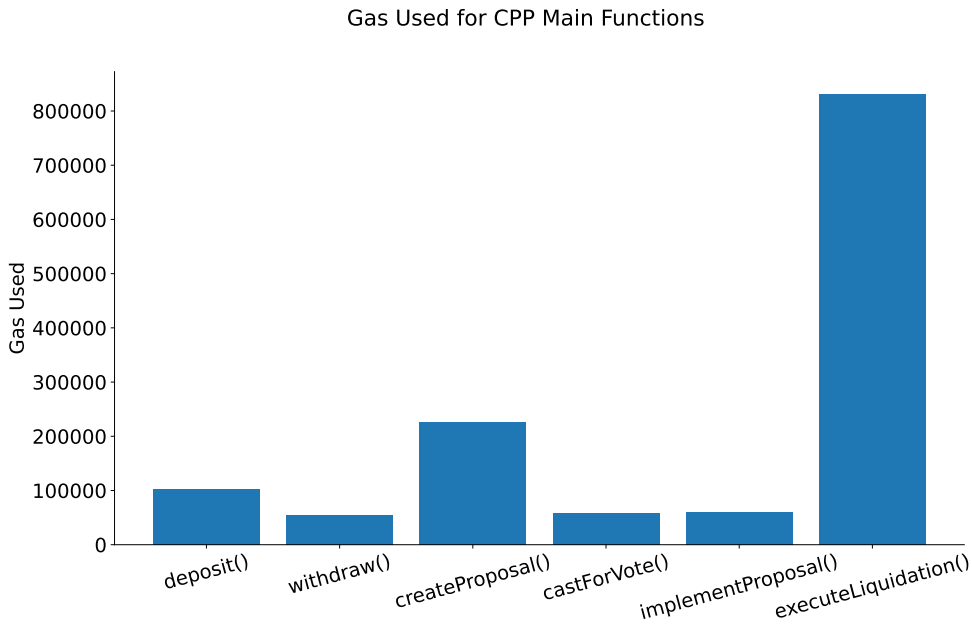

Figure 4.2: Gas used for the main CPP smart contract functions.

4.3.5 Gas Efficiency's Impact on Profitability

To demonstrate the impact of gas efficiency on the overall profitability of the liquidation system consider a lending position liquidation that yields a profit of 2260$. This average profit value is calculated from data collected by Qin et al [28] who explored liquidations over a period of two years from 2019 to 2021 across multiple Ethereum-based lending protocols. This is a conservative value since the average value for Aave liquidations appears to tend to be higher than that of most other protocols.

Testing showed that the "executeLiquidation" function in the CPP smart contract used on average 831,050 gas. According to Etherscan [15] the average gas price over the last 7 days was 21.86 Gwei. The current price for 1 Gwei is 0.0000018 USD. This means that with current network conditions calling the "executeLiquidation" function would cost:

$$\begin{aligned} \textit{Gas Cost} &= 831{,}050 \times 21.86 \times 0.0000018 \\ &= 32.70\$ \ (USD) \end{aligned}$$

This means that the system can make 69 failed loan liquidations - for any reason - in a row (on average) and still be profitable in the long run ($\lfloor 2260 \div 32.70 \rfloor = 69$).

Computational performance is vital since increased gas usage would impact the overall economical profitability of the system. Had the CPP smart contract design not employed the aforementioned techniques and used computationally expensive data structures and

loops, the gas usage would eat into the profitability of the system. Doubling the gas usage would half the number of failed loan liquidations that the system could handle and still remain profitable in the long term.

In practice, a failed loan liquidation would use less gas than discussed here due to the early exit require statements that prevent unnecessary computations from taking place. This is a worst-case scenario analysis, though, it is still a relevant metric.

4.4 Security and Decentralisation Evaluation

Security is crucial in smart contract code design. In the case of the proposed system, large amounts of valuable assets in the form of tokens are stored and utilised by the system making it an appealing target for an adversary. It is important to use relevant security techniques when designing such smart contracts since it protects against bad actors and also promotes a greater level of trust from users leading to wider adoption.

A function modifier (shown below) was utilised in the CPP smart contract to protect the "executeLiquidation" function from being called by any address that is not the address of the designated Liquidation Opportunity Oracle. This modifier effectively wraps itself around the function and ensures that not just anyone can make the smart contract attempt to liquidate a loan since if they could they would be able to waste the pool's funds by repeatedly calling the function with incorrect liquidation information so that it would fail.

```
1 modifier onlyLOO() {
2     require(msg.sender == LOO_ADDRESS);
3     _;
4 }
```

In addition to this, the smart contract validates the liquidation opportunity information passed into it on-chain to ensure that the liquidation opportunity is a valid one. It does this by fetching real-time on-chain data from the Aave protocol and performing the necessary calculations such as verifying the correct amount of debt token to liquidate. This also promotes the decentralisation of the system and reduces the power of the LOO address. Even if the LOO address somehow became malicious, it would not be able to execute fake liquidations. This is also related to the previously discussed fail-safe mechanisms that revert the transaction if conditions are not met, minimising the scale that an adversary can attack the system and providing the genuine pool participants time to make a governance proposal and alter the LOO address to a safe one.

The CPP smart contract is also security-aware when it comes to acquiring and selling tokens on the Uniswap protocol. A common problem with the Uniswap protocol is that trades can be front-run: Front-running is when an adversary exploits information about an upcoming transaction in Ethereum and carefully constructs a related trade to profit from the victim's trade. It can happen when the Uniswap protocol is not used safely and recent price data is not used to set a maximum amount to pay for a trade. To counter this vulnerability, the CPP smart contract fetches current price data on the asset that is attempting to swap so that can appropriately set the maximum amount to pay (within a small range). The smart contract fetches this value and leaves a 2% leeway to account

for any slippage during the trade but not so much as to be vulnerable to an attack. The code for this mechanism is shown below.

```
(
    uint256 priceUSDC,
    uint256 decimals
) = getOraclePrice(allowlist[debtAsset].usdPriceFeedAddress);

// Purchase the debt token via Uniswap.
uniswapAcquireDebtAsset(debtAsset, verifiedDebtToCover, (((
    verifiedDebtToCover) * uint256(priceUSDC)) / (10**
    combinedDecimals)) * 102 / 100); // 2% leeway
```

Reentrancy attacks are notorious on the Ethereum blockchain. The attack is a malicious exploit that consists of a smart contract repeatedly calling itself before it has completed prior operations and thus allowing the attacker to do things such as drain the smart contract of its tokens. This attack relates particularly to the "withdraw" function of the CPP smart contract. The code protects itself from this attack by making any state changes before transferring any funds out of the contract, this is why the internal smart contract variable that stores a user's stake in the pool is altered before the funds are actually transferred to the user. The code below details the "withdraw" function.

```
function withdraw() public {
    uint256 userWithdrawable = (poolShare[msg.sender] *
        poolMultiplier) / scalingFactor;

    // Make all state changes before transferring funds.
    poolShare[msg.sender] = 0;

    bool success = usdc.transfer(msg.sender, userWithdrawable);
    require(success, "Withdrawal was unsuccessful.");
}
```

The system makes use of a governance system to allow the stakeholders to ensure its security by agreeing on the types of assets that are safe to work with, as well as, how those assets should be treated i.e. the minimum and maximum liquidation amounts. Again, this encourages decentralised management of the system.

Finally, the general architecture of the CPP smart contract promotes decentralisation by restricting the powers of the centralised aspect of the system, namely the Liquidation Opportunity Oracle. Its functionality is clearly written in the code and various validation and fail-safe mechanisms take place to ensure that whenever it does make a call to "executeLiquidation" it cannot severely impact the system for an extended period of time if it attempts to act maliciously.

Chapter 5

Conclusion and Further Work

In this paper, a decentralised loan liquidation scheme was presented. The proposed system democratises the loan liquidation process of the Aave lending protocol, providing a technique for DeFi users to come together, pooling their funds in a trustless manner to collaboratively liquidate unhealthy lending positions and collectively earn a profit.

The design decisions of the proposed system were justified given the context and the environment in which it operates. A trade-off had to be made between the performance and viability of the scheme against the decentralisation of the whole system, in particular, the Liquidation Opportunity Oracle introduced a required level of centralisation, though, this was mitigated by the architecture and design decisions made in the Community Pool Protocol such as the governance system and the hard-coded limited powers of the LOO.

The work conducted in this paper achieves the set objectives set out in section 1.2. The system does:

- Allow DeFi users to collaborate in a trustless manner to liquidate loans on the Aave protocol by pooling their funds. This is facilitated by the design of the CPP smart contract.
- Operate in a decentralised manner, offering solutions to mitigate the level of centralisation that is required. The governance system and limited power of the LOO address help to achieve this objective.
- Explores and justifies various design choices in the context of decentralised finance. Some of the main design choices described in this paper are:
    - The best way to identify unhealthy user lending positions. The LOO program utilises Aave's subgraph on The Graph to fetch this information.
    - How to determine whether a liquidation opportunity is likely profitable or not. This is performed using estimate calculations and governance-set parameters.
    - The best way to manage users' deposited funds. The chosen solution was to store funds in USDC and use Uniswap to swap these funds for the required

tokens as they are needed.

- How to avoid unnecessary risk by performing validation checks before carrying out on-chain code execution as well as constructing the correct parameters when performing tasks such as purchasing a token on Uniswap.
- How to let stakeholders govern the system in a decentralised manner. This is achieved via the proportional governance system described in section 3.3.3.

- Implement a secure and gas-efficient smart contract design.
- Demonstrates how systems can be designed to enable DeFi users to collaborate in a trustless manner when aspects of centralisation are required. This is shown through the design decisions that handle the LOO aspect of the scheme effectively without exposing the system to a large degree of centralisation risk.

Furthermore, DeFi loan liquidations are complex and an area for further work. Some areas of further work directly related to the proposed system of this project are:

- Exploration of optimal liquidation opportunity discovery.
  - Due to activity on the Aave protocol and the price movement of many of the assets used as collateral or borrowed (therefore affecting the health of users' positions), the LOO must periodically fetch information and use it to try to identify liquidation opportunities. There is no set way to determine when and what information to fetch, thus, there is an opportunity to develop more efficient and intelligent solutions than blindly brute-force searching.
- Improvement of governance capabilities.
  - Whilst the governance system presented in this project is sufficient, it would be worthwhile to investigate the improvements that could be made by implementing a more complex system. One direction would be more granular types of votes such as emergency votes that are quick changes used in urgent times such as a malicious LOO address.
- Use of a token to represent pool participation.
  - The proposed system just internally tracks the stake of the pool participants. This could be changed to utilise a token that tracks and represents participation. Delving into this area may lead to further usage of these tokens within other protocols similar to how aTokens on Aave can be used in other protocols to earn further yield.

Overall, this paper proposes a community-run decentralised loan liquidation scheme that runs on the Ethereum blockchain allowing DeFi users to collaboratively liquidate loans on the Aave protocol without needing to understand the intricate workings of the liquidation method.

Bibliography

[1] Ethereum API | IPFS API & Gateway | ETH Nodes as a Service. URL: https://infura.io.

[2] Ethers.js Documentation. URL: https://docs.ethers.org/v5/.

[3] Hardhat | Ethereum development environment for professionals by Nomic Foundation. URL: https://hardhat.org.

[4] Mocha - the fun, simple, flexible JavaScript test framework. URL: https://mochajs.org/.

[5] Aave. Aave Protocol v2 Whitepaper, December 2020. URL: https://github.com/aave/protocol-v2/blob/ce53c4a8c8620125063168620eba0a8a92854eb8/aave-v2-whitepaper.pdf.

[6] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 Core. *Uniswap Labs, Tech. Rep*, March 2021. URL: https://uniswap.org/whitepaper-v3.pdf.

[7] Andreas A. Aigner and Gurvinder Dhaliwal. UNISWAP: Impermanent Loss and Risk Profile of a Liquidity Provider, June 2021. URL: https://papers.ssrn.com/abstract=3872531, doi:10.2139/ssrn.3872531.

[8] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An Analysis of Uniswap markets. *Cryptoeconomic Systems*, 0(1), April 2021. URL: https://cryptoeconomicsystems.pubpub.org/pub/angeris-uniswap-analysis/release/15, doi:10.21428/58320208.c9738e64.

[9] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. SoK: Lending Pools in Decentralized Finance. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, Lecture Notes in Computer Science, pages 553–578, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-63958-0_40.

[10] Lorenz Breidenbach, Christian Cachin, Alex Coventry, Steve Ellis, Ari Juels, Andrew Miller, Brendan Magauran, Sergey Nazarov, Alexandru Topliceanu, Fan

Zhang, Benedict Chan, Farinaz Koushanfar, Daniel Moroz, and Florian Tramer. Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks. April 2021. URL: https://research.chain.link/whitepaper-v2.pdf.

[11] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014. URL: https://finpedia.vn/wp-content/uploads/2022/02/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf.

[12] CoinMarketCap. Cryptocurrency Prices, Charts And Market Capitalizations. URL: https://coinmarketcap.com/.

[13] DefiLlama. Aave v2 Protocol Data. URL: https://defillama.com/protocol/aave-v2.

[14] DefiLlama. Uniswap Protocol Data. URL: https://defillama.com/dexs/uniswap.

[15] etherscan.io. Ethereum (ETH) Blockchain Explorer. URL: http://etherscan.io/.

[16] Emilio Frangella and Lasse Herskind. Aave V3 Technical Paper. January 2022. original-date: 2021-07-27T08:54:40Z. URL: https://github.com/aave/aave-v3-core.

[17] Hasu. Understanding Automated Market-Makers, Part 1: Price Impact, April 2021. URL: https://research.paradigm.xyz/amm-price-impact.

[18] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. Risks and Returns of Uniswap V3 Liquidity Providers, September 2022. arXiv:2205.08904 [q-fin]. URL: http://arxiv.org/abs/2205.08904, doi:10.1145/3558535.3559772.

[19] Stuart Hoegner. Affidavit 4–30. April 2019. URL: http://media.kalzumeus.com/tether-docs/bitfinex-response-to-nyag.pdf.

[20] Uniswap Labs. What is Price Impact? URL: https://support.uniswap.org/hc/en-us/articles/8671539602317-What-is-Price-Impact-.

[21] Cheng Lim, TJ Saw, and Calum Sargeant. Smart Contracts: Bridging the Gap Between Expectation and Reality, July 2016. URL: https://blogs.law.ox.ac.uk/business-law-blog/blog/2016/07/smart-contracts-bridging-gap-between-expectation-and-reality.

[22] Qinwei Lin, Chao Li, Xifeng Zhao, and Xianhai Chen. Measuring Decentralization in Bitcoin and Ethereum using Multiple Metrics and Granularities. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pages 80–87, April 2021. ISSN: 2473-3490. doi:10.1109/ICDEW53142.2021.00022.

[23] Amani Moin, Kevin Sekniqi, and Emin Gun Sirer. SoK: A Classification Framework for Stablecoin Designs. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020 , Kota Kinabalu, Malaysia, February 10–14,*

2020 Revised Selected Papers, pages 174–197, Berlin, Heidelberg, February 2020. Springer-Verlag. doi:10.1007/978-3-030-51280-4_11.

[24] Jude Molloy and Xiao Chen. A Decentralised Loan Liquidation Scheme Basedon an Ethereum-enabled DeFi System. *UKPEW2022*, December 2022. URL: https://www.researchgate.net/publication/366185584_UKPEW2022.

[25] Emily Nicolle. Why tether and stablecoin USDT have become a big crypto worry, January 2023. Section: financialservices. URL: https://www.afr.com/companies/financial-services/why-tether-and-stablecoin-usdt-have-become-a-big-crypto-worry-20230103-p5c

[26] Daniel Perez, Sam M. Werner, Jiahua Xu, and Benjamin Livshits. Liquidations: DeFi on a Knife-Edge. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 457–476, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-662-64331-0_24.

[27] Jithil Ponnan. ERC-20 Token Standard, January 2023. URL: https://ethereum.org.

[28] Kaihua Qin, Liyi Zhou, Pablo Gamito, Philipp Jovanovic, and Arthur Gervais. An empirical study of DeFi liquidations: incentives, risks, and instabilities. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, pages 336–350, New York, NY, USA, November 2021. Association for Computing Machinery. doi:10.1145/3487552.3487811.

[29] Haseeb Qureshi. Stablecoins: designing a price-stable cryptocurrency | HackerNoon, February 2018. URL: https://hackernoon.com/stablecoins-designing-a-price-stable-cryptocurrency-6bf24e2689e5.

[30] Yaniv Tal, Brandon Ramirez, and Jannis Pohlmann. The Graph: A Decentralized Query Protocol for Blockchains. March 2018. original-date: 2018-03-20T23:03:37Z. URL: https://github.com/graphprotocol/research/blob/c35ff4832ad0306bc9cea3720599751ed263e894/papers/whitepaper/the-graph-whitepaper.pdf.

[31] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. *ACM Computing Surveys*, 55(11):238:1–238:50, February 2023. doi:10.1145/3570639.

[32] Abdullah A. Zarir, Gustavo A. Oliva, Zhen M. (Jack) Jiang, and Ahmed E. Hassan. Developing Cost-Effective Blockchain-Powered Applications: A Case Study of the Gas Usage of Smart Contract Transactions in the Ethereum Blockchain Platform. *ACM Transactions on Software Engineering and Methodology*, 30(3):1–38, July 2021. URL: https://dl.acm.org/doi/10.1145/3431726, doi:10.1145/3431726.