Credit Score-Based Lending System On Ethereum Platform

A Project

Presented to

The Faculty of the Department of Computer Science San

José State University

In Partial Fulfillment

of the Requirements for the Degree Master

of Science

by

Mayuri Shimpi

May 2024

The Designated Project Committee Approves the Project Titled

Credit Score-Based Lending System On Ethereum Platform

by

Mayuri Shimpi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2024

ABSTRACT

Credit Score-Based Lending System On the Ethereum Platform by

Mayuri Shimpi

Traditional banking systems act as intermediaries, assessing risks and profiting from interest rate differentials. Credit scores, provided by trusted bureaus, are commonly used to evaluate the creditworthiness of borrowers. Cryptocurrencies have emerged as a significant and innovative medium due to their decentralized nature, operating without reliance on a central authority, such as a government.

This report describes a project to implement the Autonomous Lending system on the Ethereum Platform (ALOE), as proposed in [1], aiming to seamlessly integrate traditional credit scoring methodologies for evaluating a borrower's risk of default. The objective of this project report is to establish a robust understanding of cryptocurrencies and the Ethereum platform and describe the implementation of pivotal components of the ALOE system, as presented by Austin, Potika, and Pollett in 2023. Specifically, the report aims to incorporate essential functionalities of the Credit Bureau Smart Contract (CBSC). This entails the creation of a Notary tasked with verifying borrowers using real-world FICO scores, SSNs, and Ethereum Addresses. The Notary further divides the user's identity among various auditors and invokes the initializeLedger function to establish a credit score for the borrower. The CBSC plays a pivotal role in connecting lenders and borrowers. Finally, in cases of loan repayment failure, lenders have the option to engage auditors to disclose the client's identity.

Keywords: DeFi, Blockchain, Ethereum, Smart Contracts

ACKNOWLEDGMENTS

I'd like to extend my heartfelt thanks to Professor Chris Pollett for his unwavering support, patience, and insightful feedback. His guidance has been pivotal to my growth over the past two semesters. Our weekly brainstorming and whiteboarding sessions have significantly contributed to the enhancement and refinement of my project. His consistent availability to answer my questions and steer me in the right direction has been invaluable.

I also want to acknowledge the Department of Computer Science at SJSU for offering comprehensive courses and resources that were instrumental in accomplishing my project goals.

My gratitude extends to my committee members and all faculty for sharing their knowledge and expertise. Additionally, I'm grateful to my peers and family for their unwavering support and encouragement throughout this journey.

TABLE OF CONTENTS

I. Introduction

In the fast-changing field of financial technology, new cryptocurrencies such as Ethereum have offered game-changing features beyond simple digital transactions. At the heart of this breakthrough are smart contracts, which are the self-executing contracts with the agreement's terms explicitly put into code. These smart contracts are more than just theoretical constructions; they are strong instruments that can algorithmically execute and verify a wide range of transactions, profoundly altering how we think about financial agreements.

Money lending is an excellent example of a real-world application that will profit greatly from this technology. Historically, this process has been thoroughly embedded in a system in which banks act as mediators. Banks play an important role in moving cash from depositors to borrowers while benefitting from the interest rate disparity. They methodically examine borrowers' trustworthiness using credit scores, which are numerical representations that assist them gauge the likelihood of payback and, as a result, the risk and profitability of a loan.

Even though this conventional banking paradigm has been helpful to us for a long time, it has drawbacks. Dependence on centralized organizations may result in inefficiencies, higher expenses, and restricted accessibility, particularly for people without collateral or credit histories. This is the domain of blockchain technology and systems such as the Ethereum Platform's Autonomous Lending System (ALOE), which was suggested in [1].

ALOE, which is based on the Ethereum blockchain, eliminates the need for traditional banks by facilitating direct, unsecured fund borrowing between people or organizations. Through the utilization of blockchain technology's openness, security, and efficiency, ALOE presents a decentralized option that is inventive and comprehensive.

This project's primary goal is to thoroughly investigate the Autonomous Lending System on the

Ethereum Platform (ALOE), as it was suggested in [1]. Understanding its unique qualities and functions as well as how ALOE handles identity verification in a transparent and safe manner will be the main topics of discussion. Furthermore, the project intends to investigate how ALOE facilitates seamless interactions between borrowers and lenders on a decentralized platform and assesses borrower creditworthiness through on-chain credit scoring. The order of this project report is as follows: The preparatory work and research completed during the project's first half, as well as a quick summary of the findings, are covered in the next chapter. The relevant work in the DeFi and loan systems is then covered in the following chapter.

Following related works, the system architecture is explained along with the implementation strategies used. The results and experiments chapter then overviews the unit testing and other observations drafted from the functioning lending system. Lastly, the future scope of the project has been mentioned along with the concluding remarks.

II. Related work

This chapter delineates various conventional peer-to-peer lending platforms alongside blockchain-based systems such as Banksocial.

Conventional peer-to-peer (P2P) lending sites, such as LendingClub (2007), Prosper Marketplace (2006), and Zopa (2005), rely on users to post personal information and loan amounts. The platform evaluates creditworthiness, sets an interest rate, and makes the loan available for funding to investors. Interest on loans is earned by investors, and origination and service fees are profitable for P2P platforms. P2P platforms purchase loans that are first issued by banks and then take over the collection process in the event of defaults.

Still, difficulties are present. Problems with loan criteria not meeting investor expectations have been documented; examples include LendingClub in 2016 and Prosper Marketplace in 2008. Furthermore, borrowers have experienced adverse interest rates as a result of misdated loans; this was most noticeable at LendingClub in 2016. These difficulties emphasize the demand for loan alternatives that are more efficient and transparent

Using blockchain technology, decentralized finance, or DeFi, refers to financial services that aim to reconstruct traditional financial systems in a decentralized fashion. The Decentralized Autonomous Organization (DAO) is a major participant in this ecology. A decentralized autonomous organization (DAO) is essentially a type of cryptocurrency that is frequently referred to as the DAO coin. Specifically, it can purchase DAO coins using other cryptocurrencies and activate voting processes. It runs on a system of smart contracts.

To cast votes on different issues, members of a DAO utilize its native currencies, called governance tokens. They might include choices about how to allocate funds or the organization's strategic orientation. These monies frequently consist of the money used to buy DAO currencies in the context of blockchain-based lending.

Some well-known DAOs in the DeFi space include:

MakerDAO (2017)	Known for its stablecoin DAO, which is backed by collateral in the form of other cryptocurrencies.
Compound (2018)	A lending platform where users can lend and borrow various cryptocurrencies.
Aave (2020)	Offers lending and borrowing services with features like flash loans and interest rate swaps.

Table 1 Some existing DAOs

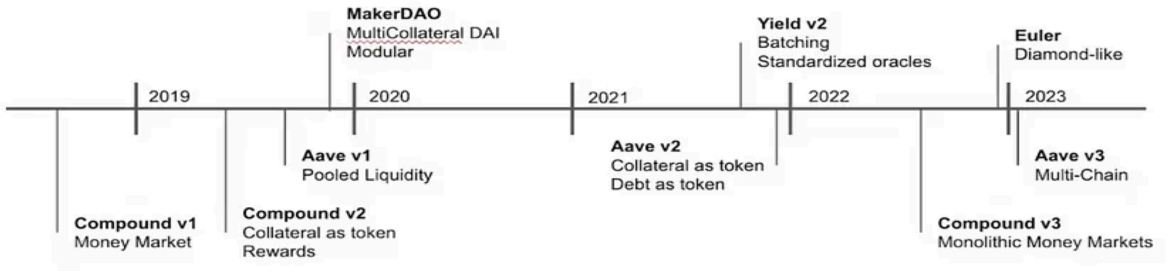


Figure 1: Evolution of Lending on Ethereum

While DAOs are a novel way to lend money, some lenders built on blockchain, such as

2022-founded Banksocial, provide more conventional loan services.

Banksocial enables investors to buy coins in the platform's cryptocurrency, facilitating loans that are both secured and unsecured. Loans are then financed with the money produced from these investments. The credit assessment algorithm of Banksocial sets the terms of these loans. Investors get a passive income stream in the form of interest payments, which are contingent on the performance of these loans.

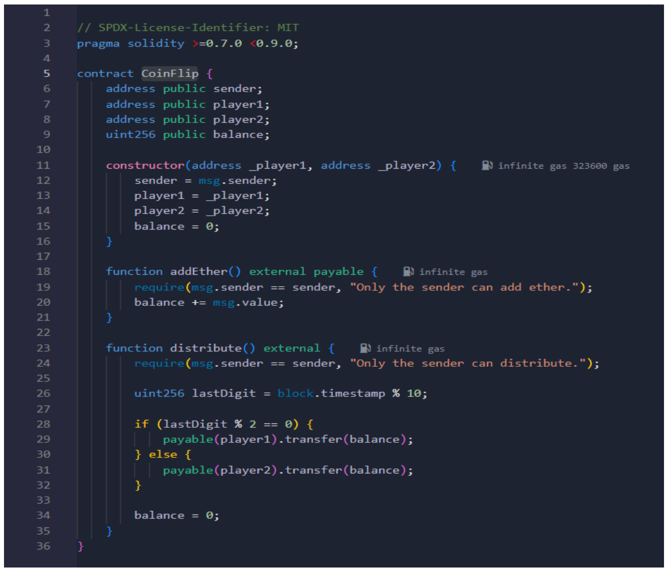
The initial success of overcollateralized borrowing was demonstrated by Ethereum apps such as Compound, Aave, and MakerDAO [14]. Subsequent versions of these systems included features like yield farming, composability, and pooled liquidity as they gained popularity. These improvements were made in an effort to increase market share and capitalize on favorable market conditions, particularly during bullish trends.

III. Preliminary works

The project's initial phase was mainly focused on research work and trying to implement a few essential components of the system. This chapter briefly describes the work carried out during the first half of the project, along with its applications and benefits to the outcome. The chapter includes a description of the basic implementation of a smart contract, the concept of decentralization that is central to the blockchain, the credit score calculations, and the local setup of the blockchain.

3.1 Test and deploy smart contacts on Remix IDE

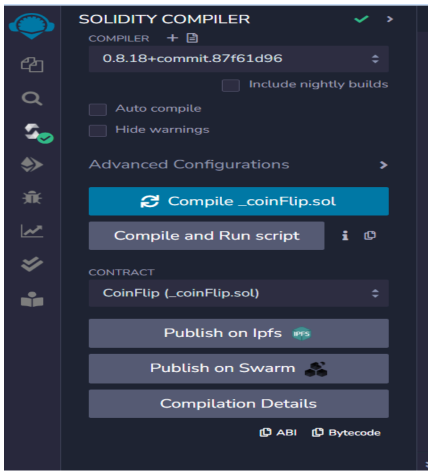
Web3.js stands as a pivotal JavaScript library in the realm of blockchain development, specifically designed to facilitate interactions with the Ethereum blockchain. It's extensive feature set enables developers to build solid and dynamic decentralized apps (dApps) that take advantage of the Ethereum blockchain's potential. Remix IDE is an online integrated development environment that provides a full suite of tools to make creating, evaluating, and implementing smart contracts easier.



```

// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract CoinFlip {
    address public sender;
    address public player1;
    address public player2;
    uint256 public balance;

    constructor(address _player1, address _player2) {    infinite gas 323600 gas
        sender = msg.sender;
        player1 = _player1;
        player2 = _player2;
        balance = 0;
    }

    function addEther() external payable {    infinite gas
        require(msg.sender == sender, "Only the sender can add ether.");
        balance += msg.value;
    }

    function distribute() external {    infinite gas
        require(msg.sender == sender, "Only the sender can distribute.");

        uint256 lastDigit = block.timestamp % 10;

        if (lastDigit % 2 == 0) {
            payable(player1).transfer(balance);
        } else {
            payable(player2).transfer(balance);
        }

        balance = 0;
    }
}
```

Figure 2: CoinFlip Smart Contract

A sender can start a basic coin-flipping game between two players, player1 and player2, with the use of the CoinFlip smart contract. It uses the Ethereum blockchain to build a simple coin-flipping game in which two participants can join by adding ether to the contract. The final digit of the block timestamp determines the distribution of the ether balance, introducing a random component to the gameplay. Despite having a straightforward design, it shows how smart contracts can run decentralized apps.


SOLIDITY COMPILER

COMPILER

0.8.18+commit.87f61d96

Include nightly builds

Auto compile

Hide warnings

Advanced Configurations

Compile _coinFlip.sol

Compile and Run script

CONTRACT

CoinFlip (_coinFlip.sol)

Publish on Ipfs

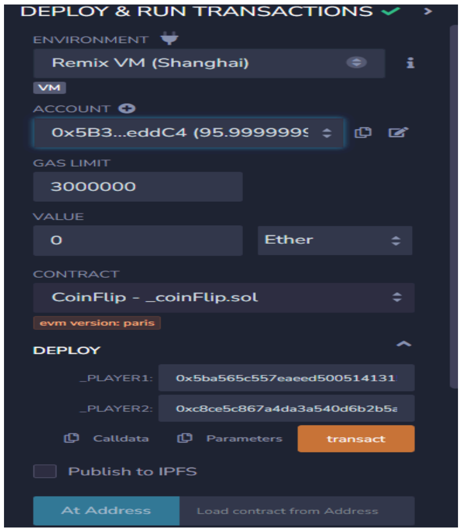
Publish on Swarm

Compilation Details

ABI

Bytecode

Figure 3: Compile and deploy the smart contract


DEPLOY & RUN TRANSACTIONS

ENVIRONMENT

Remix VM (Shanghai)

VM

ACCOUNT

0x5B3...eddC4 (95.999999

GAS LIMIT

3000000

VALUE

0 Ether

CONTRACT

CoinFlip - _coinFlip.sol

evm version: paris

DEPLOY

_PLAYER1: 0x5ba565c557eaeed500514131

_PLAYER2: 0xc8ce5c867a4da3a540d6b2b5a

Calldata

Parameters

transact

Publish to IPFS

At Address

Load contract from Address

Figure 4: Deploy the smart contract

Choosing the contract, entering the necessary parameters (Sender, player1, player2), and hitting the Transact button to start deployment in the selected test environment are the simple steps involved in deploying a smart contract on Remix IDE. By using this method, developers may ensure functionality and security prior to real-world deployment by testing and validating their smart contracts in a simulated environment before releasing them to the Ethereum mainnet.

3.2 Decentralization

The idea of decentralization is crucial to the architecture and tenets of Bitcoin, according to Satoshi Nakamoto's groundbreaking essay "Bitcoin: A Peer-to-Peer Electronic Cash System" [2]. The lack of a central body or middleman in charge of the Bitcoin network and transactions is referred to in this sense as decentralization.

The blockchain, a decentralized ledger, is introduced in this paper. All Bitcoin transactions are transparently and permanently recorded on this ledger. It is preserved by the nodes in the network using a consensus process known as Proof-of-Work (PoW). The decentralization of the Bitcoin system is further enhanced by this consensus mechanism, which guarantees that transactions are verified and added to the blockchain in a decentralized and secure manner.

Unlike traditional financial systems, where the currency is controlled and regulated by a central authority, such as banks or governments, Bitcoin is decentralized. The network jointly manages the creation of new Bitcoins and transaction validation, ensuring the currency system's decentralization and democratization.

Additionally, decentralization improves the Bitcoin network's resilience and security. By dispersing control and decision-making across the network, Bitcoin becomes more resistant to attacks, censorship, and system failure. This distributed architecture protects Bitcoin's integrity and continuity, giving it a strong and safe alternative to established financial systems.

Although the main use of Bitcoin is as a digital money or store of value, it is not as capable as Ethereum when it comes to smart contracts. Vitalik Buterin proposed Ethereum at the end of 2013, and the platform was introduced in 2015 with an expanded goal [3]. Ethereum was intended to be a decentralized platform where programmers could create and implement smart contracts and decentralized applications (dApps). In contrast to Bitcoin, which functions primarily as virtual money, Ethereum acts as a flexible framework for a range of blockchain-related uses. Ethereum first employed a PoW consensus algorithm akin to that of Bitcoin. With the release of Ethereum 2.0, it has started to switch to a Proof-of-Stake (PoS) consensus process that uses less energy. Compared to Bitcoin, Ethereum's programming language is more programmable and adaptable, enabling developers to design a wide range of functionalities, including token creation, decentralized finance (DeFi) applications, non-fungible tokens (NFTs), and much more.

3.3 Credit Score Calculation

Ensuring that organizations are accurately identified is a crucial component of credit scores. Preventing situations where a single entity—a person or a company—can pose as several different entities is crucial. Furthermore, safeguards need to be put in place to stop bad actors from ruining the credit histories of well-behaved people. By preserving trust and equity for all parties involved, this guarantees the accuracy and dependability of credit scoring systems.

The FICO Score 8, which is frequently used for house loans, is covered in the reference paper. It was created by the Fair Isaac Corporation (now FICO) [1]. Payment history, debt burden, length of credit history, types of credit used, and recent credit searches make up its five foundational elements. A weighted score formula, which is written as follows, is used to integrate these elements:

$$WS = 0.35 \text{ X } PH + 0.3 \text{ X } DB + 0.15 \text{ X } LoCH + 0.1 \text{ X } ToC + 0.1 \text{ X } RCS$$

The resulting WS is then transformed using a proprietary function to produce a final credit score between 350 and 850, reflecting the probability of non-default.

The computational simplicity of the WS formula, involving only addition and multiplication, makes it suitable for homomorphic encryption. The Paillier cryptosystem is specifically highlighted for its cost-effectiveness in terms of computational resources. This encryption method allows for operations to be performed on encrypted data, maintaining borrower privacy on public blockchains.

The project's implementation, inspired by [1], includes a model in the smart contract that tracks two main ratios motivated by the component of FICO 8 as well as a set of window ratios, and additional bookkeeping information is maintained for incremental updates.

The Underpay Ratio (UPR) measures the ratio of time that the amount an individual owes is more than the amount that they should owe versus the total period of all loans that individual has ever carried. It captures an individual’s payment history as a single number, calculated as TOT divided by the period from the first loan to the present.

The Current Debt Burden Ratio (CDBR) is a metric used to assess an individual's current level of debt relative to their average debt burden over time. It is calculated by dividing the current outstanding debt (COB) by the sum of the average outstanding debt (AOB) and three "pseudo" standard deviations.

3.4 Smart Contract Using Solidity

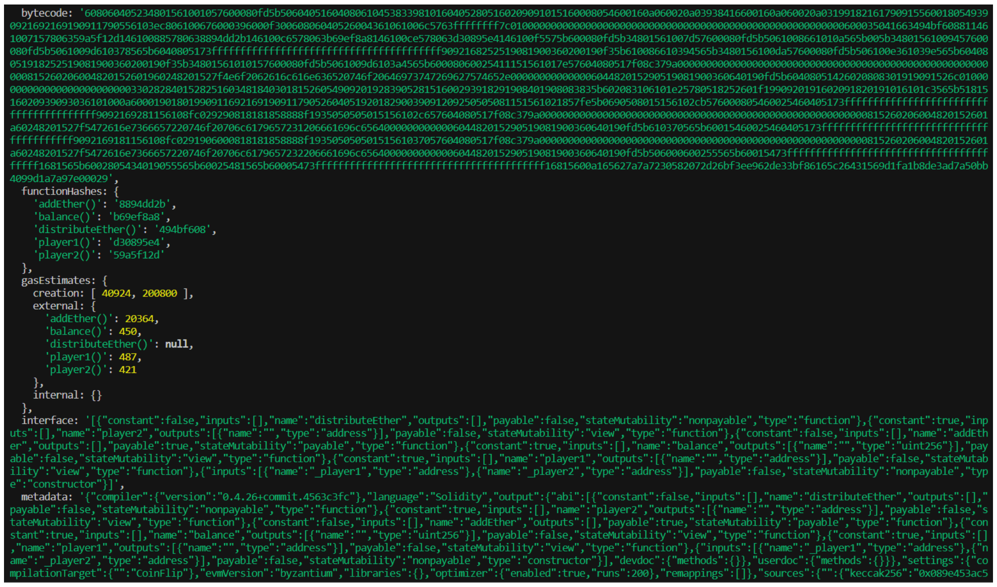
Setting up a local blockchain environment is crucial, particularly during the development and testing phases of blockchain applications. The Web3.js JavaScript library was utilized to provide an interface for interacting with the Ethereum blockchain. This library enables developers to build applications capable of interacting with smart contracts, querying blockchain data, and sending transactions.

The *_coinFlip.sol* file was compiled to test and deploy the smart contract. The essential properties obtained from the compiled file are the Application Binary Interface (ABI) and the bytecode. The bytecode is deployed to the blockchain to facilitate the smart contract's functionality, while the ABI serves as a human-readable map of the bytecode.

To compile the Solidity code, the 'solc' Solidity compiler was installed. In the project's root directory, the command npm install --save solc was executed. Subsequently, a compile.js file was created in the root directory to handle the compilation process. Node.js modules such as path and fs, along with the solc compiler, were used in the *compile.js* file to read and compile the *_coinFlip.sol* file.

The source variable in the compile.js file contains the raw Solidity code from the *_coinFlip.sol* file. The compile() function was then invoked on this source code, specifying that a single contract is being compiled. This process outputs the compiled bytecode and ABI for the CoinFlip smart contract.

Executing node compile.js in the terminal displays the compiled bytecode and ABI, providing the necessary components to deploy and interact with the CoinFlip smart contract on the Ethereum blockchain.



```
  bytecode: '608060405234801561001057600080fd5b506040516040806104538339810160405280516020909101516000805460016 0a060020a03938416600160a060020a0319918216179091556001805493 9
0921692169190911790556103ec806100676000396000f3006080604052600436106100 6c5763ffffffff7c0100000000000000000000000000000000000000000000000000000000600035041663494bf60881146
1007157806359a5f12d146100885780638894dd2b146100c6578063b69ef8a8146100ce578063d30895e4146100f5575b600080fd5b34801561007d57600080fd5b5061008661010a565b005b34801561009457600
080fd5b5061009d610378565b6040805173ffffffffffffffffffffffffffffffffffffffff90921682525190819003602001 90f35b61008661039456 5b3480156100da57600080fd5b506100e361039e565b60408
05191825251908190036020019 0f35b34801561010157600080fd5b5061009d6103a4565b60008060025411151561017e57604080517f08c379a000000000000000000000000000000000000000000000000000000
000815260206004820152601960248201527f4e6f2062616c616e636520746f20646973747269627574652e000000000000006044820152905190819003606 40190fd5b6040805142602080830191909152 6c01000
00000000000000000000330282840152825160348184030181526054909201928390528151600293918291908401908083835b602083106101e25780518252601f1990920191602091820191016101c3565b51815
16020939093036101000a6000190180199091169216919091179052604051920182900390912092505050811515 61021857fe5b06905080151561 02cb576000805460025460405173ffffffffffffffffffffffff
ffffffffffffffff9092169281156108fc0292908181818588 88f1935050505015156102c657604080517f08c379a000000000000000000000000000000000000000000000000000000000815260206004820152601
a60248201527f5472616e7366657220746f20706c6179657231206661696c656400000000000060448201529051908190036064 0190fd5b610370565b60015460025460405173ffffffffffffffffffffffffffff
ffffffffffff9092169181156108fc0291906000818181858888f1935050505015156103705760408 0517f08c379a000000000000000000000000000000000000000000000000000000000815260206004820152601
a60248201527f5472616e7366657220746f20706c6179657232206661696c6564000000000000604482015290519081900360640190fd5b50600060025556 5b60015473ffffffffffffffffffffffffffffffffffff
fffff1681565b600280543401905556 5b60025481565b60005473ffffffffffffffffffffffffffffffffffffffff16815600a165627a7a7230582072d26bf3ee962de33bf86165c26431569d1fa1b8de3ad7a50bb
4099d1a7a97e00029',
  functionHashes: {
    'addEther()': '8894dd2b',
    'balance()': 'b69ef8a8',
    'distributeEther()': '494bf608',
    'player1()': 'd30895e4',
    'player2()': '59a5f12d'
  },
  gasEstimates: {
    creation: [ 40924, 200800 ],
    external: {
      'addEther()': 20364,
      'balance()': 450,
      'distributeEther()': null,
      'player1()': 487,
      'player2()': 421
    },
    internal: {}
  },
  interface: '[{"constant":false,"inputs":[],"name":"distributeEther","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inp
uts":[],"name":"player2","outputs":[{"name":"","type":"address"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[],"name":"addEth
er","outputs":[],"payable":true,"stateMutability":"payable","type":"function"},{"constant":true,"inputs":[],"name":"balance","outputs":[{"name":"","type":"uint256"}],"pay
able":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[],"name":"player1","outputs":[{"name":"","type":"address"}],"payable":false,"stateMutab
ility":"view","type":"function"},{"inputs":[{"name":"_player1","type":"address"},{"name":"_player2","type":"address"}],"payable":false,"stateMutability":"nonpayable","typ
e":"constructor"}]',
  metadata: '{"compiler":{"version":"0.4.26+commit.4563c3fc"},"language":"Solidity","output":{"abi":[{"constant":false,"inputs":[],"name":"distributeEther","outputs":[],"
payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"player2","outputs":[{"name":"","type":"address"}],"payable":false,"s
tateMutability":"view","type":"function"},{"constant":false,"inputs":[],"name":"addEther","outputs":[],"payable":true,"stateMutability":"payable","type":"function"},{"con
stant":true,"inputs":[],"name":"balance","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[]
,"name":"player1","outputs":[{"name":"","type":"address"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs":[{"name":"_player1","type":"address"},{"n
ame":"_player2","type":"address"}],"payable":false,"stateMutability":"nonpayable","type":"constructor"}],"devdoc":{"methods":{}},"userdoc":{"methods":{}}},"settings":{"co
mpilationTarget":{"":"CoinFlip"},"evmVersion":"byzantium","libraries":{},"optimizer":{"enabled":true,"runs":200},"remappings":[]},"sources":{"":{"keccak256":"0x089e453ac5
```

Figure 5: Compiling a smart contract using the ‘solc’ compiler

The smart contract was tested using the Mocha Testing framework, a feature-rich JavaScript test framework compatible with both Node.js and browser environments. Mocha facilitates asynchronous testing and provides flexible and accurate reporting of test results.

In addition to Mocha, Ganache and Web3 were utilized for testing. Ganache offers a set of unlocked accounts for local testing, eliminating the need for public or private keys to access them. Web3 provides libraries for interacting with Ethereum nodes, and for testing, a local node provided by Ganache was connected.

IV Architecture and implementation

This system is based on the paper by Thomas Austin, et. al. from [1] and is developed within the Ethereum blockchain ecosystem, leveraging its smart contract functionality executed through its virtual machine and developed using Solidity as the high-level language. Within Ethereum, two primary account types exist:

- Externally Owned Accounts (EOA), owned by external users and associated with public keys
- Contract Accounts, which execute specific code.

This architecture comprises four core processes:

- Borrower Registration
- Loan Creation
- Borrowing and Lending
- Credit Score calculation

These processes are designed to interact within the Ethereum blockchain environment, ensuring secure and transparent execution of lending operations while leveraging the decentralized nature of blockchain technology.

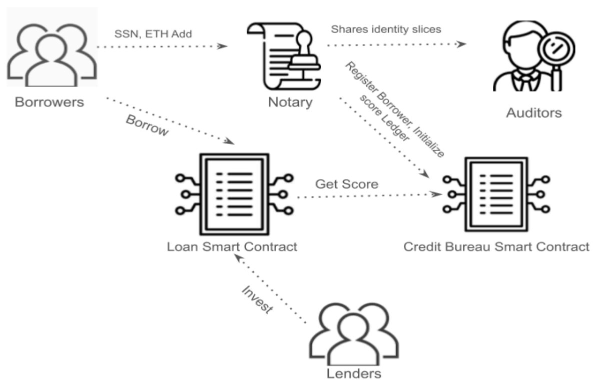


Figure 6: Core processes of the Lending system

The architecture of the project involves a borrower registering with a notary in the real world, who verifies their information and interacts with the Credit Bureau Smart Contract (CBSC). The CBSC handles tasks such as verifying unused addresses, initializing score ledgers, and facilitating loan contracts. Auditors receive shares of identity, i.e., the Ethereum address of the Borrower. Loan smart contracts enable borrowers to borrow and lenders to invest, with funds managed securely until loan conditions are met. Borrowers interact with loan contracts through methods like ‘isReady()’ and ‘makePayment()’, updating credit scores via the CBSC.

4.1 Borrower Registration

During the registration phase, the client provides their real-world identity, such as their social security number (SSN), to the credit bureau. This process involves several entities. This section includes the architecture of the Borrower Registration and its implementation.

4.1.1 Architecture

The Borrower Registration phase consists of the following entities along with their purpose:

- Borrower: A real person with a unique SSN and an Ethereum address.
- Notary: A trusted entity that validates the user's identity, credit score, and Ethereum address. It divides the user's identity among auditors and initializes a credit score for the borrower in the credit bureau smart contract.
- Auditors: Responsible for storing shares of the borrower's real-world identity. They ensure the confidentiality of their share of the secret.
- Credit Bureau Smart Contract (CBSC): Stores a mapping between Ethereum addresses and associated credit scoring information. It connects lenders and borrowers, tracks loans and repayments, and updates the borrower's credit score. The notary is the only

entity authorized to invoke certain methods of this smart contract.

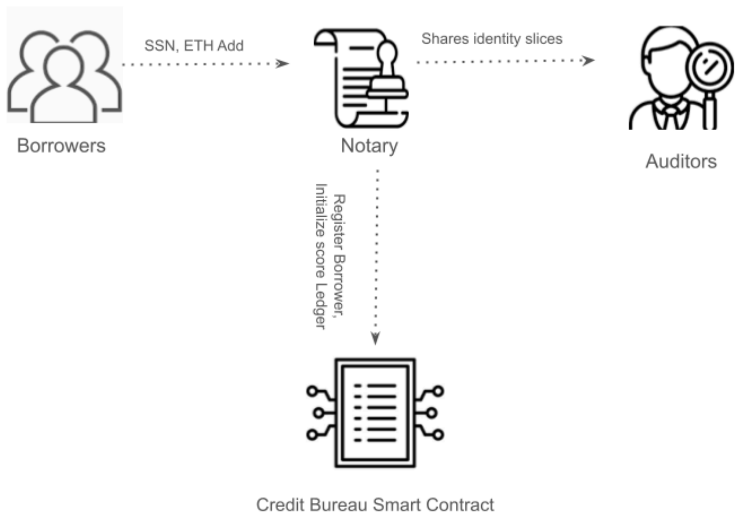
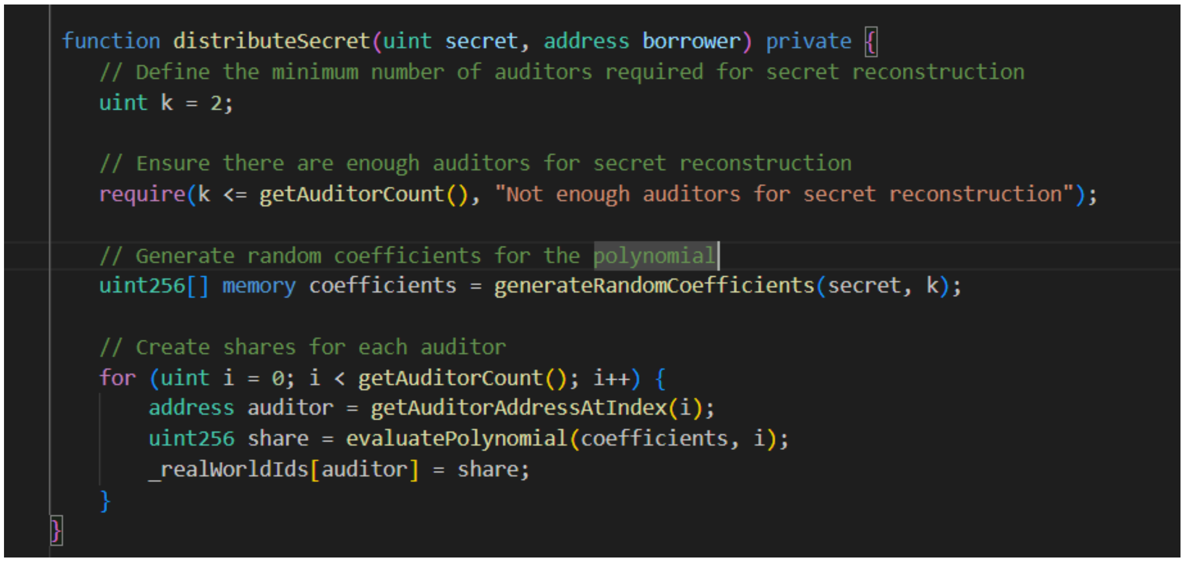


Figure 7: Borrower Registration

Significant trust is placed in the notary during the registration phase. If not performed accurately, a borrower's credit score may be misrepresented, compromising anonymity. The auditors collectively tie a user's Ethereum address to their real-world identity, but no individual auditor can do so. The registration process involves the borrower providing their SSN and Ethereum address to the notary for verification, followed by the notary initiating the registration process in the CBSC and distributing the borrower's SSN among auditors using secret sharing schemes. Finally, the notary registers the borrower with an initial credit score in the CBSC.

4.1.2 Implementation

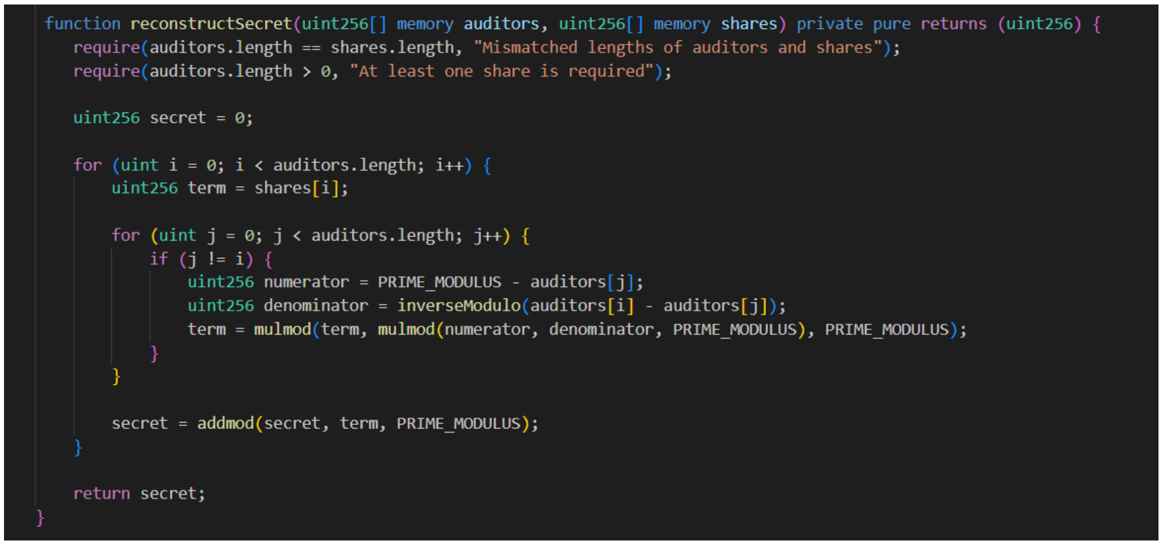
The Solidity code for the Register smart contract implements a secret sharing algorithm, drawing inspiration from Shamir's secret sharing algorithm as proposed in the referenced paper [1]. Specifically, the distribute secret function utilizes a polynomial secret-sharing scheme inspired by Shamir's algorithm. This scheme divides the borrower's SSN into shares distributed among auditors, ensuring no single auditor has access to the complete SSN. By generating random coefficients and evaluating the polynomial at each auditor's index, the algorithm effectively distributes the secret while preserving its integrity.



```
function distributeSecret(uint secret, address borrower) private {
   // Define the minimum number of auditors required for secret reconstruction
   uint k = 2;

   // Ensure there are enough auditors for secret reconstruction
   require(k <= getAuditorCount(), "Not enough auditors for secret reconstruction");

   // Generate random coefficients for the polynomial
   uint256[] memory coefficients = generateRandomCoefficients(secret, k);

   // Create shares for each auditor
   for (uint i = 0; i < getAuditorCount(); i++) {
       address auditor = getAuditorAddressAtIndex(i);
       uint256 share = evaluatePolynomial(coefficients, i);
       _realWorldIds[auditor] = share;
   }
}
```

Figure 8: Distribute Identity slice

The ‘reconstructSecret’ function then enables the reconstruction of the original secret from these shares, leveraging principles from Shamir's algorithm to maintain data privacy and confidentiality. This implementation underscores the practical implications and applications of Shamir's secret sharing algorithm within blockchain-based systems, enhancing security and privacy in sensitive data management. In case of a loan default, the system implements a ReconstructSecret function aimed at revealing the borrower's identity.



```
function reconstructSecret(uint256[] memory auditors, uint256[] memory shares) private pure returns (uint256) {
    require(auditors.length == shares.length, "Mismatched lengths of auditors and shares");
    require(auditors.length > 0, "At least one share is required");

    uint256 secret = 0;

    for (uint i = 0; i < auditors.length; i++) {
        uint256 term = shares[i];

        for (uint j = 0; j < auditors.length; j++) {
            if (j != i) {
                uint256 numerator = PRIME_MODULUS - auditors[j];
                uint256 denominator = inverseModulo(auditors[i] - auditors[j]);
                term = mulmod(term, mulmod(numerator, denominator, PRIME_MODULUS), PRIME_MODULUS);
            }
        }

        secret = addmod(secret, term, PRIME_MODULUS);
    }

    return secret;
}
```

Figure 9: Reconstruct Borrower

4.2 Loan Smart Contract

Initiating a new loan must be done by an Externally Owned Account (EOA), which assumes the role of the loan creator and defines the loan terms. As this process involves ether from the creator, they can charge a fee from borrower interest. If extra ether accrues due to an early lender fund claim, the additional interest also benefits the loan creator.

4.2.1 Architecture

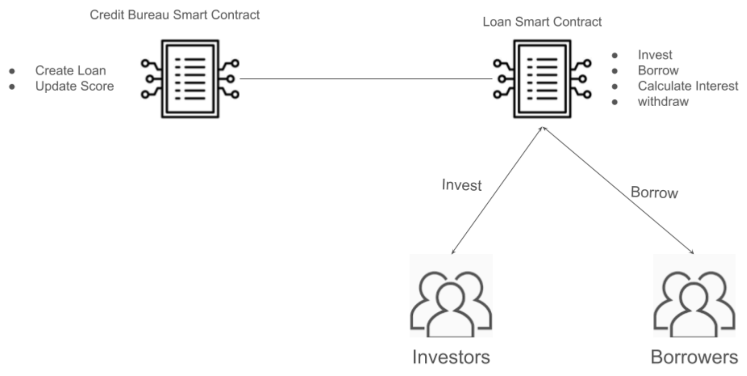


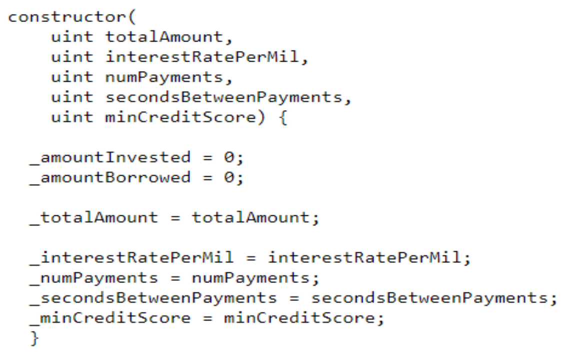
Figure 10: Loan Smart Contract

At the center is the "Loan Smart Contract" (LoanSC), which serves as the primary component responsible for managing loans. It offers functionalities such as investing funds, borrowing loans, calculating interest, and processing withdrawals.

4.2.2 Implementation

This section presents the implementation of the Loan Smart Contract, facilitating connections between lenders and borrowers. The Credit Bureau Smart Contract initializes loans for the association by borrowers and lenders. Lenders invest funds, borrowers specify their needs, and interest rates are calculated based on terms set by the initiating EOA. Borrowers can make payments and monitor credit scores, with updates to the Credit Bureau smart contract reflecting changes. Lenders can also withdraw funds.

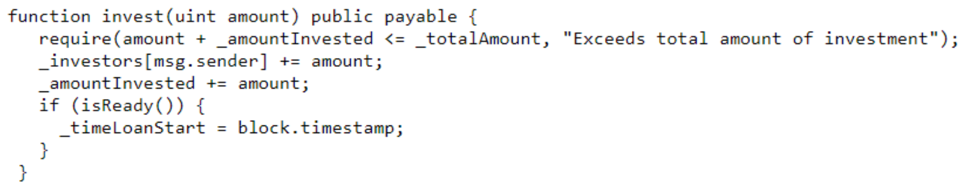
The constructor initializes crucial variables for the Loan smart contract, defining parameters such as the total loan amount, interest rate, number of payments, time intervals between payments, and minimum credit score requirement. Additionally, it sets ‘_amountInvested’ and ‘_amountBorrowed’ to 0, which serve to track investment and borrowing activities within the contract. These initializations establish the framework for managing loan transactions and ensure accurate tracking of financial interactions between lenders and borrowers.



```
constructor(
    uint totalAmount,
    uint interestRatePerMil,
    uint numPayments,
    uint secondsBetweenPayments,
    uint minCreditScore) {

  _amountInvested = 0;
  _amountBorrowed = 0;

  _totalAmount = totalAmount;

  _interestRatePerMil = interestRatePerMil;
  _numPayments = numPayments;
  _secondsBetweenPayments = secondsBetweenPayments;
  _minCreditScore = minCreditScore;
  }
```

Figure 11: Loan Smart Contract: Constructor

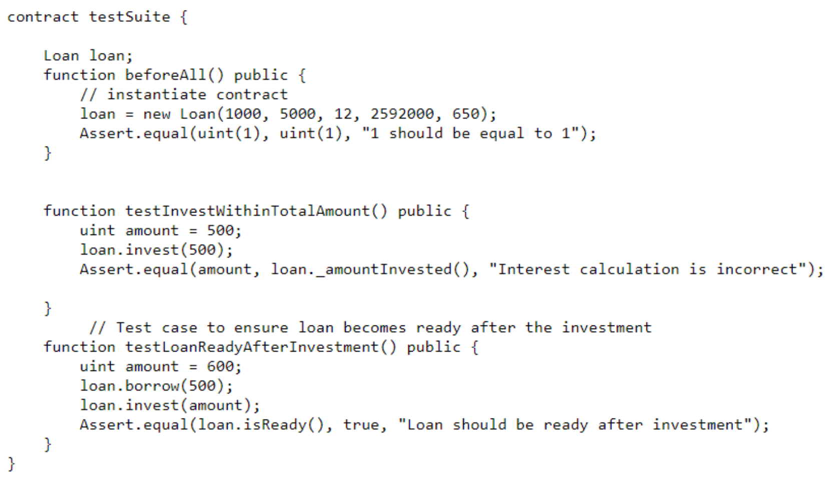
The *invest()* function allows lenders to deposit funds into the Loan smart contract. It withdraws the specified amount from the lender's account and adds it to the contract balance. If enough funds are received, the transaction timestamp marks the start of the loan. This timestamp is used to calculate accrued interest, ensuring accurate interest calculations based on the loan's initiation time.



```
function invest(uint amount) public payable {
   require(amount + _amountInvested <= _totalAmount, "Exceeds total amount of investment");
   _investors[msg.sender] += amount;
   _amountInvested += amount;
   if (isReady()) {
     _timeLoanStart = block.timestamp;
   }
 }
```

Figure 12: Loan Smart Contract: Invest function

Testing the *invest()* function requires evaluating different scenarios to confirm its accuracy. This includes ensuring the investment amount is within the allowed total and verifying the loan's readiness after the investment.

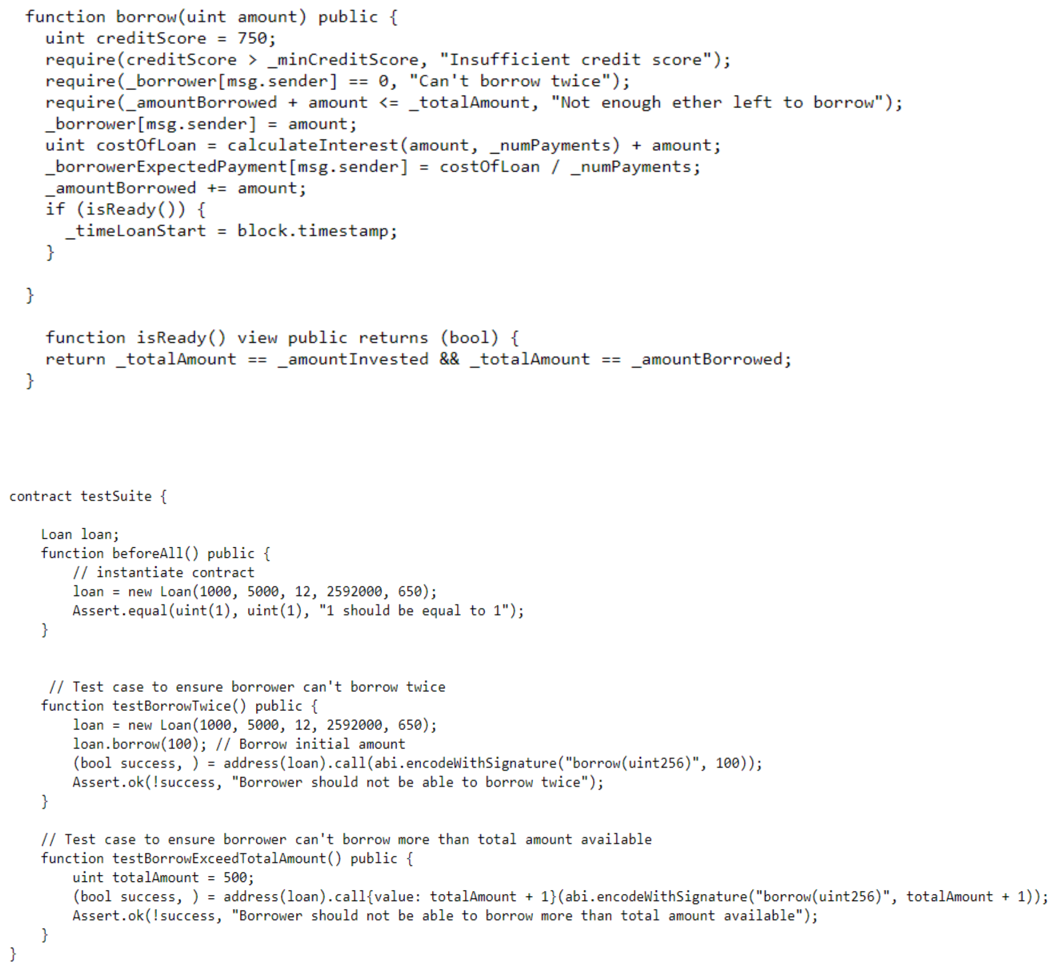


```
contract testSuite {

    Loan loan;
    function beforeAll() public {
        // instantiate contract
        loan = new Loan(1000, 5000, 12, 2592000, 650);
        Assert.equal(uint(1), uint(1), "1 should be equal to 1");
    }


    function testInvestWithinTotalAmount() public {
        uint amount = 500;
        loan.invest(500);
        Assert.equal(amount, loan._amountInvested(), "Interest calculation is incorrect");

    }
         // Test case to ensure loan becomes ready after the investment
    function testLoanReadyAfterInvestment() public {
        uint amount = 600;
        loan.borrow(500);
        loan.invest(amount);
        Assert.equal(loan.isReady(), true, "Loan should be ready after investment");
    }
}
```

Figure 13: Loan Smart Contract: Testing the Invest function

The *borrow()* function allows the borrower (msg.sender) to request a loan, contingent upon meeting the loan requirements including the borrower's credit score and the requested amount. It involves a call to *getScore()* which may incur gas costs. Additionally, the function checks whether the loan has already been funded to prevent unnecessary recomputations, potentially caching previous results. The *isReady()* function determines if the loan has been fully funded.

Testing the *borrow(uint amount)* function entails examining different scenarios to ascertain its accuracy. This includes ensuring that a borrower cannot borrow twice and verifying that the borrower cannot request an amount exceeding the total available amount.

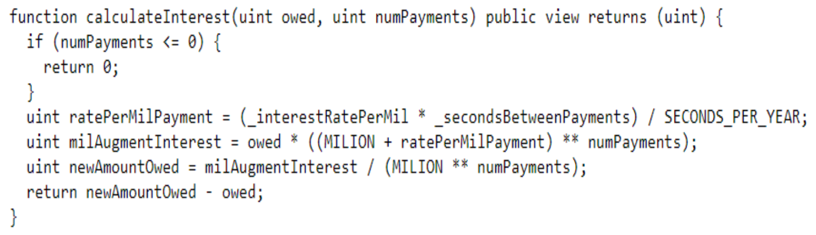


```
function borrow(uint amount) public {
  uint creditScore = 750;
  require(creditScore > _minCreditScore, "Insufficient credit score");
  require(_borrower[msg.sender] == 0, "Can't borrow twice");
  require(_amountBorrowed + amount <= _totalAmount, "Not enough ether left to borrow");
  _borrower[msg.sender] = amount;
  uint costOfLoan = calculateInterest(amount, _numPayments) + amount;
  _borrowerExpectedPayment[msg.sender] = costOfLoan / _numPayments;
  _amountBorrowed += amount;
  if (isReady()) {
    _timeLoanStart = block.timestamp;
  }

}

  function isReady() view public returns (bool) {
  return _totalAmount == _amountInvested && _totalAmount == _amountBorrowed;
}
```

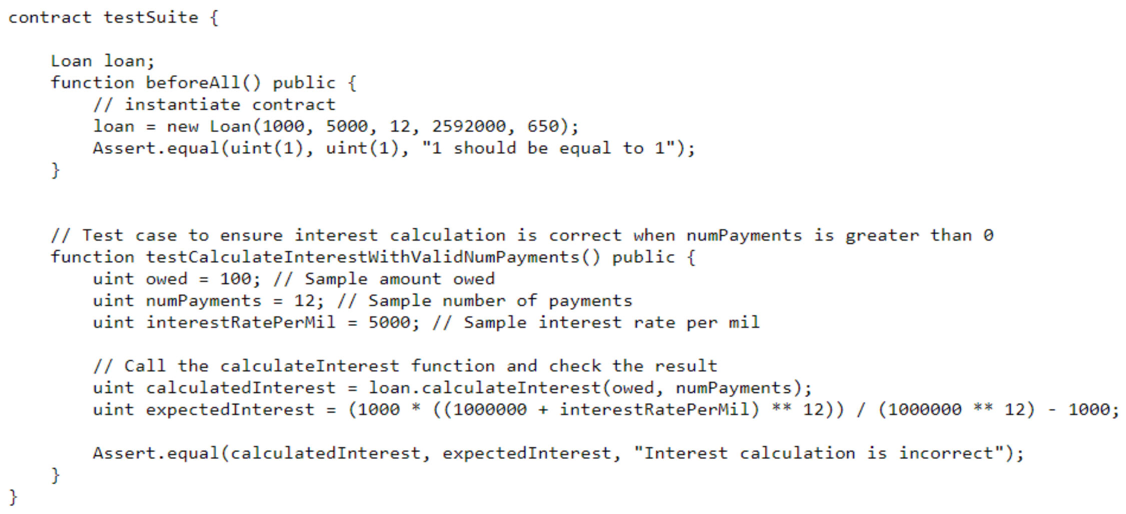
```
contract testSuite {

    Loan loan;
    function beforeAll() public {
        // instantiate contract
        loan = new Loan(1000, 5000, 12, 2592000, 650);
        Assert.equal(uint(1), uint(1), "1 should be equal to 1");
    }


     // Test case to ensure borrower can't borrow twice
    function testBorrowTwice() public {
        loan = new Loan(1000, 5000, 12, 2592000, 650);
        loan.borrow(100); // Borrow initial amount
        (bool success, ) = address(loan).call(abi.encodeWithSignature("borrow(uint256)", 100));
        Assert.ok(!success, "Borrower should not be able to borrow twice");
    }

    // Test case to ensure borrower can't borrow more than total amount available
    function testBorrowExceedTotalAmount() public {
        uint totalAmount = 500;
        (bool success, ) = address(loan).call{value: totalAmount + 1}(abi.encodeWithSignature("borrow(uint256)", totalAmount + 1));
        Assert.ok(!success, "Borrower should not be able to borrow more than total amount available");
    }
}
```

Figure 14: Loan Smart Contract: Borrow function

The *calculateInterest(uint owed, uint numPayments)* function computes the accrued interest on a loan based on the provided parameters. It first checks if the number of payments is greater than zero. Then, it calculates the interest rate per payment interval and uses it to determine the augmented interest. This augmented interest is then used to calculate the new amount owed after all payments. Finally, it returns the difference between the new amount owed and the original amount owed, representing the accrued interest on the loan.



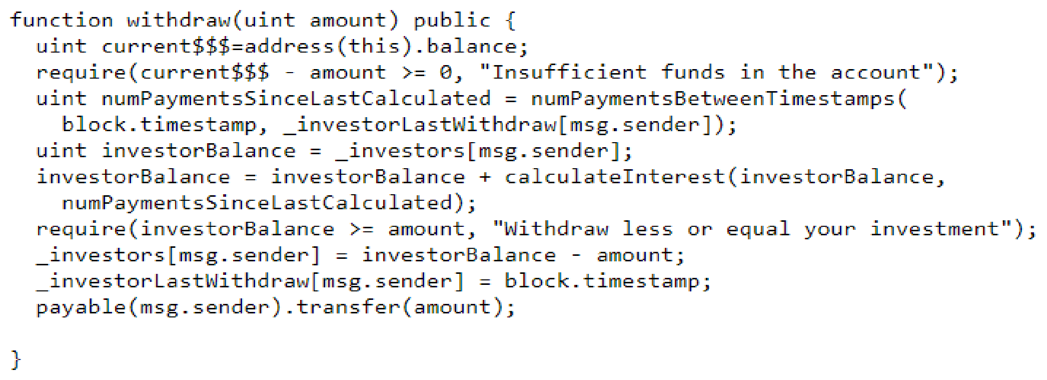
```
function calculateInterest(uint owed, uint numPayments) public view returns (uint) {
  if (numPayments <= 0) {
    return 0;
  }
  uint ratePerMilPayment = (_interestRatePerMil * _secondsBetweenPayments) / SECONDS_PER_YEAR;
  uint milAugmentInterest = owed * ((MILION + ratePerMilPayment) ** numPayments);
  uint newAmountOwed = milAugmentInterest / (MILION ** numPayments);
  return newAmountOwed - owed;
}
```



```
contract testSuite {

    Loan loan;
    function beforeAll() public {
        // instantiate contract
        loan = new Loan(1000, 5000, 12, 2592000, 650);
        Assert.equal(uint(1), uint(1), "1 should be equal to 1");
    }

    // Test case to ensure interest calculation is correct when numPayments is greater than 0
    function testCalculateInterestWithValidNumPayments() public {
        uint owed = 100; // Sample amount owed
        uint numPayments = 12; // Sample number of payments
        uint interestRatePerMil = 5000; // Sample interest rate per mil

        // Call the calculateInterest function and check the result
        uint calculatedInterest = loan.calculateInterest(owed, numPayments);
        uint expectedInterest = (1000 * ((1000000 + interestRatePerMil) ** 12)) / (1000000 ** 12) - 1000;

        Assert.equal(calculatedInterest, expectedInterest, "Interest calculation is incorrect");
    }
}
```

Figure 15: Loan Smart Contract: Calculate Interest function

The withdraw(uint amount) function enables an investor to withdraw a specified amount of ether from the contract. It first checks if the contract balance is sufficient to fulfill the withdrawal request. Then, it calculates the number of payments made since the last withdrawal and updates the investor's balance by adding the accrued interest. After ensuring that the investor's balance is adequate for withdrawal, it deducts the withdrawn amount from the investor's balance and updates the timestamp of the last withdrawal. Finally, it transfers the specified amount of ether to the investor's address.



```
function withdraw(uint amount) public {
  uint current$$$=address(this).balance;
  require(current$$$ - amount >= 0, "Insufficient funds in the account");
  uint numPaymentsSinceLastCalculated = numPaymentsBetweenTimestamps(
    block.timestamp, _investorLastWithdraw[msg.sender]);
  uint investorBalance = _investors[msg.sender];
  investorBalance = investorBalance + calculateInterest(investorBalance,
    numPaymentsSinceLastCalculated);
  require(investorBalance >= amount, "Withdraw less or equal your investment");
  _investors[msg.sender] = investorBalance - amount;
  _investorLastWithdraw[msg.sender] = block.timestamp;
  payable(msg.sender).transfer(amount);

}
```

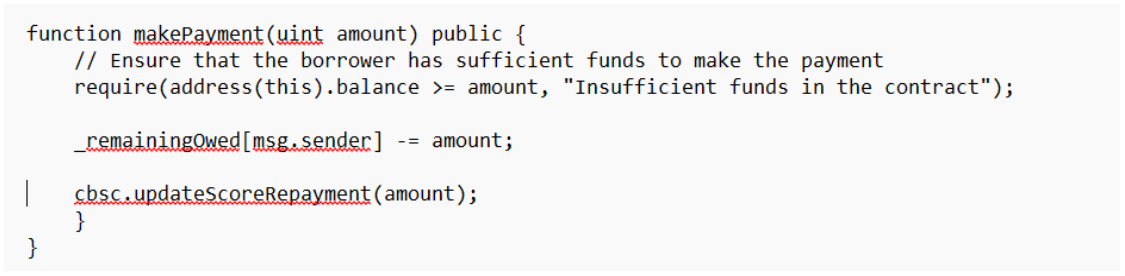
Figure 16: Loan Smart Contract: Withdraw function

These functions and some helper functions like *isReady()* (returns whether or not the loan has been funded) conclude the implementation and unit testing of the Loan Smart Contract. Next, I will continue to examine the procedure of a borrower acquiring a loan and subsequently repaying it as defined in the ALOE system [1]. We presume that the loan has already been established but has not yet received full funding or gathered the required number of borrowers. To monitor credit scores, we incorporate the following functions into the CreditBureau smart contract. The function updateScoreBorrow notifies the credit bureau of a loan being issued for a specified amount. Similarly, the function updateScoreRepayment informs the Credit Bureau smart contract that a loan has received a repayment payment of a certain amount. Furthermore, the function getScore retrieves the credit score of a client based on the given parameters such as the requested amount, interest rate per mil, number of payments, and seconds between payments.

4.3 Borrowing and Lending

The Credit Bureau Smart Contract's (CBSC) *findLoan* method, though not currently implemented in the project, is planned for future integration. This method will enable borrowers and lenders to search for Loan smart contracts based on specific criteria. Borrowers will still call the borrow(uint amount) method, and lenders will use the *invest* method, while the *findLoan* method will be added later to facilitate loan discovery.

Borrowers initiate the borrow(uint amount) method, which involves verifying their credit score via the CBSC's getScore method. Lenders utilize the *invest* method to transfer funds to the Loan smart contract, where access to funds is limited until the fund_date. Upon reaching the *totalAmount* threshold, triggering loan funding, the timestamp is employed to compute the owed interest. Borrowers employ *isReady()* to ascertain loan readiness, *get$$$* to retrieve funds, and *makePayment* to repay loans, both of which prompt credit score updates in the Credit Bureau Smart Contract.



```
function makePayment(uint amount) public {
    // Ensure that the borrower has sufficient funds to make the payment
    require(address(this).balance >= amount, "Insufficient funds in the contract");

    _remainingOwed[msg.sender] -= amount;

    cbsc.updateScoreRepayment(amount);
    }
}
```

Figure 17: Borrower: Make Payment

After the fund_date, lenders can withdraw funds from the Loan smart contract using the withdraw(uint amount) function, limited to the smaller of the invested amount plus interest or the current balance of the contract. The owed amount to an investor is updated with each withdrawal, considering accrued interest since the previous withdrawal or the loan fund time if none. If the owed amount reaches 0, lenders can no longer receive funds. In case of loan default, an external process can trigger a CreditBureau method with the Ethereum address involved. The CreditBureau verifies the default, and contacts auditors with proof, who respond with user identity slices. Lenders can then retrieve these slices from the CreditBureau to pursue collection off-chain. The implementation includes the *distributeSecret* and *reconstructSecret* functions for this purpose. These functions enable CreditBureau to distribute proof of default to auditors and collect user identity slices in response. They can be extended further to accommodate additional features or requirements in the future.

4.4 Credit Scoring System

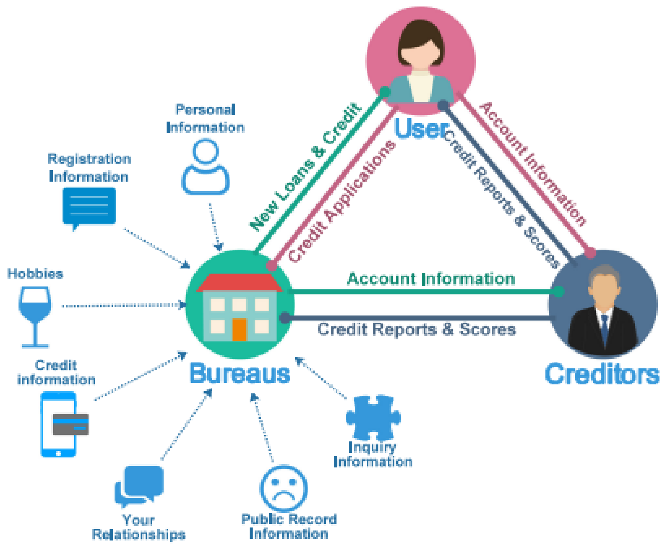


Figure 18: General Diagram of Credit Scoring

The credit system typically involves three main participants: users, credit bureaus, and creditors. Users engage in credit and loan activities, which are reported to credit bureaus. These bureaus are responsible for collecting, recording, and distributing credit data, such as new accounts, balances, inquiries, and payment history. Creditors then request this credit data to compute the user's credit score, which serves as a reference for trust assessment. Credit scores are generated using algorithms analyzing credit data, with factors like payment history, owed amounts, credit history length, new credit, and credit mix influencing the final score. For instance, FICO scores consider payment history (35%), amounts owed (30%), credit history length (15%), new credit (10%), and credit mix (10%).

4.4.1 Architecture

The credit scoring system operates on-chain, enabling transparent and verifiable credit scoring procedures. For each Ethereum borrower address, the system maintains and updates a set of ratios, including Underpay Ratio (UPR), Current Debt Burden Ratio (CDBR), and Current Payment Burden Ratio (CPBR), which are stored in the smart contract using integer pairs to avoid Ethereum dust. These ratios are inspired by the components detailed in the ALOE system[1].

4.4.2 Coordinates and Ratios

The UPR represents the ratio of time a borrower's owed amount exceeds the expected amount compared to their total loan duration. On the other hand, the CDBR calculates the ratio of current outstanding debt to the average outstanding debt plus three "pseudo" standard deviations [1].

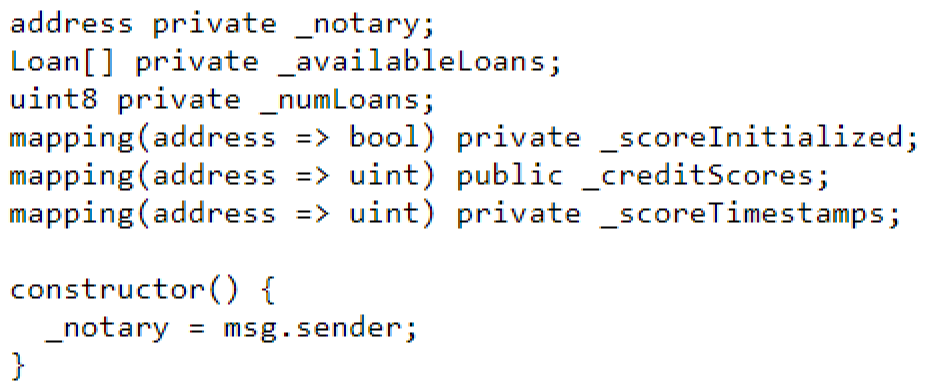
To illustrate incremental updates, let's consider the UPR. The Credit Bureau Smart Contract (CBSC) stores the timestamp of a borrower's first loan, FLT, and determines the current time, NOW. With each payment made, the CBSC updates TOT, the total time the borrower owed more than expected, and calculates the UPR as TOT divided by NOW minus FLT. For borrowers with no credit history, a loan amount below a certain threshold, and a payment below another threshold, the getScore function returns the original FICO score associated with the account divided by 1000 [1].

The chosen k-nearest neighbor implementation involves storing points in a hash table indexed by the borrower's address. This ensures that updating a score during payment is an efficient O(1) operation, minimizing gas costs. However, retrieving a score requires a linear scan of the table, which can be more costly. To keep costs comparable to real-world credit scoring, the number of points should remain relatively low.

4.4.3 Implementation

The Credit Bureau Smart Contract (CBSC) maintains a mapping linking Ethereum addresses with corresponding credit scoring data. This supplementary data includes the credit score linked with the hashed identity, a timestamp marking when this linkage was established, and a position in our credit scoring spectrum. The CBSC facilitates the connection between lenders and borrowers, monitors loan transactions and repayments, and manages updates to the borrower's credit score. Exclusive authority to invoke specific methods of this smart contract rests with the notary.

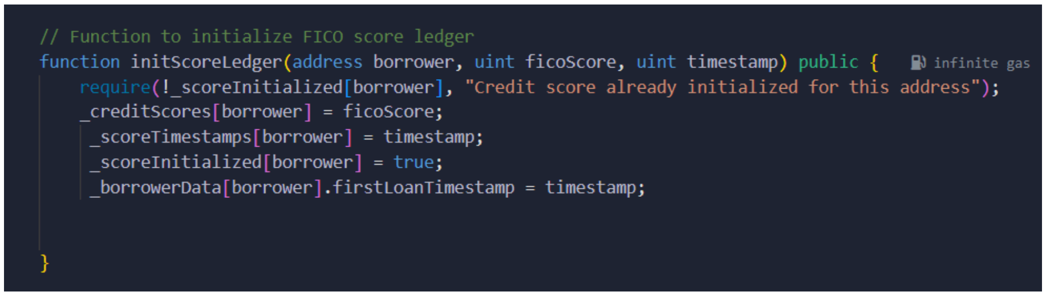
The Credit Bureau Smart Contract (CBSC) includes fields to store the notary's address, available loans, number of loans, credit score initialization status, credit scores mapped to addresses, and timestamps of score associations. The constructor initializes the notary address upon deployment.



```
address private _notary;
Loan[] private _availableLoans;
uint8 private _numLoans;
mapping(address => bool) private _scoreInitialized;
mapping(address => uint) public _creditScores;
mapping(address => uint) private _scoreTimestamps;

constructor() {
  _notary = msg.sender;
}
```

Figure 19: CBSC: Constructor

This function, *initScoreLedger*, is a public function in the CreditBureau smart contract. Its purpose is to initialize the FICO score ledger for a specific borrower. It checks if the credit score for the given borrower has not already been initialized. If it has been initialized, the function reverts with an error message.



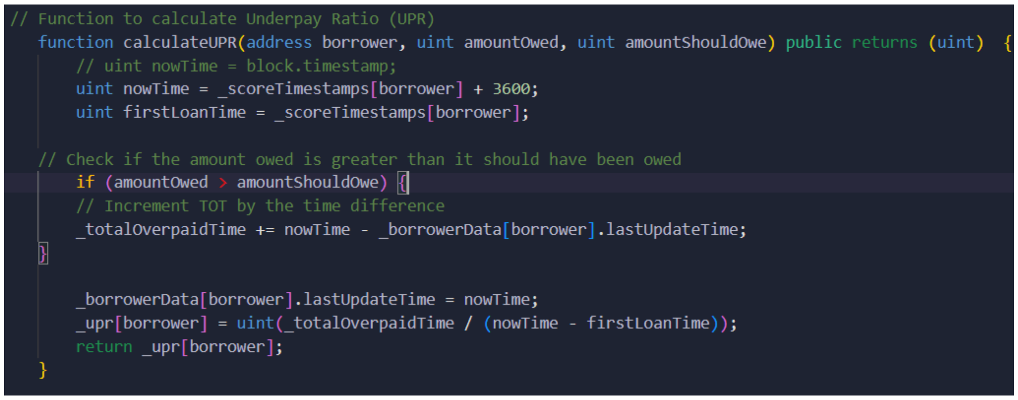
```
// Function to initialize FICO score ledger
function initScoreLedger(address borrower, uint ficoScore, uint timestamp) public {
    require(!_scoreInitialized[borrower], "Credit score already initialized for this address");
    _creditScores[borrower] = ficoScore;
     _scoreTimestamps[borrower] = timestamp;
     _scoreInitialized[borrower] = true;
     _borrowerData[borrower].firstLoanTimestamp = timestamp;


}
```

Figure 20: CBSC: Initialize Ledger

The *calculateUPR* function within the CreditBureau smart contract serves the purpose of determining the Underpay Ratio (UPR) for a specific borrower. This function takes three parameters: the borrower's Ethereum address (borrower), the current amount owed (*amountOwed*), and the expected amount that the borrower should owe (amountShouldOwe). Upon invocation, the function retrieves the current timestamp and the timestamp of the borrower's first loan from the _scoreTimestamps mapping. It then increments the current timestamp by 3600 seconds (1 hour) to simulate the passage of time since the last update.

Subsequently, the function calculates the difference in time between the current time and the time of the borrower's first loan, representing the total time span of all loans carried by the borrower.

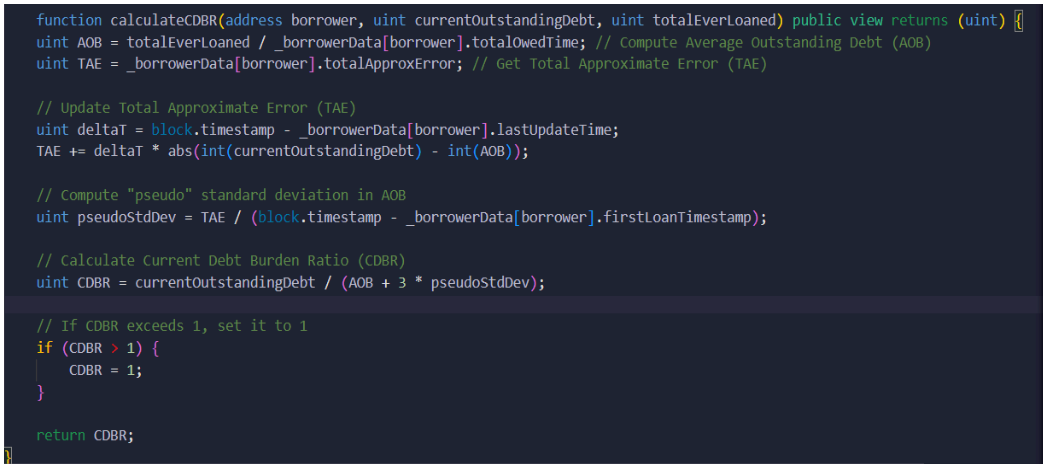
To compute the UPR, the function checks if the current amount owed exceeds the expected amount. If so, it increments the total overpaid time by the difference in time since the last update. After updating the last update time of the borrower's data to the current time, the function calculates the UPR by dividing the total overpaid time by the total time span of all loans carried by the borrower. The resulting UPR is then stored in the _upr mapping for the borrower, and the function returns this calculated UPR value.



```
// Function to calculate Underpay Ratio (UPR)
   function calculateUPR(address borrower, uint amountOwed, uint amountShouldOwe) public returns (uint)  {
       // uint nowTime = block.timestamp;
       uint nowTime = _scoreTimestamps[borrower] + 3600;
       uint firstLoanTime = _scoreTimestamps[borrower];

   // Check if the amount owed is greater than it should have been owed
       if (amountOwed > amountShouldOwe) {
       // Increment TOT by the time difference
       _totalOverpaidTime += nowTime - _borrowerData[borrower].lastUpdateTime;
   }

       _borrowerData[borrower].lastUpdateTime = nowTime;
       _upr[borrower] = uint(_totalOverpaidTime / (nowTime - firstLoanTime));
       return _upr[borrower];
   }
```

Figure 21: CBSC: Calculate UPR

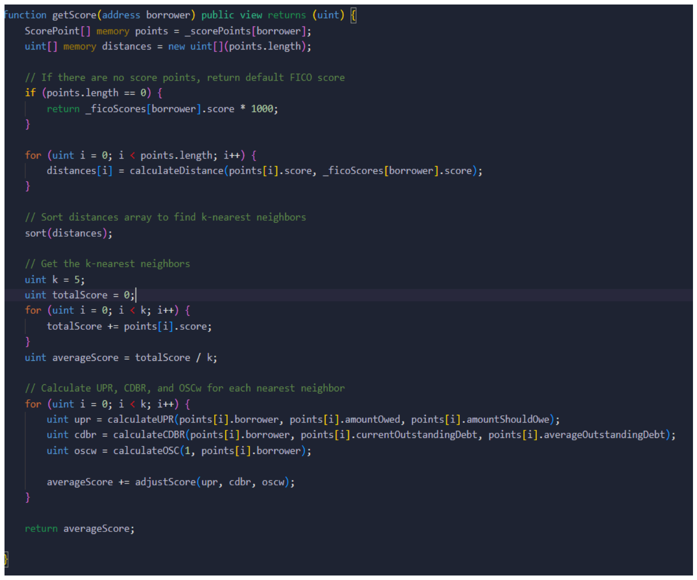
The following function calculates the Current Debt Burden Ratio (CDBR) for a borrower based on their current outstanding debt and the total ever loaned. It computes the average outstanding debt (AOB) and "pseudo" standard deviation in AOB, considering the Total Approximate Error (TAE). Finally, it computes the CDBR and adjusts it if it exceeds 1.



```
function calculateCDBR(address borrower, uint currentOutstandingDebt, uint totalEverLoaned) public view returns (uint) {
    uint AOB = totalEverLoaned / _borrowerData[borrower].totalOwedTime; // Compute Average Outstanding Debt (AOB)
    uint TAE = _borrowerData[borrower].totalApproxError; // Get Total Approximate Error (TAE)

    // Update Total Approximate Error (TAE)
    uint deltaT = block.timestamp - _borrowerData[borrower].lastUpdateTime;
    TAE += deltaT * abs(int(currentOutstandingDebt) - int(AOB));

    // Compute "pseudo" standard deviation in AOB
    uint pseudoStdDev = TAE / (block.timestamp - _borrowerData[borrower].firstLoanTimestamp);

    // Calculate Current Debt Burden Ratio (CDBR)
    uint CDBR = currentOutstandingDebt / (AOB + 3 * pseudoStdDev);

    // If CDBR exceeds 1, set it to 1
    if (CDBR > 1) {
        CDBR = 1;
    }

    return CDBR;
}
```

Figure 22: CBSC: Calculate CDBR

Next, in the getScore function, we examine the k nearest neighbors based on their underpay ratio and credit debt burden ratio, along with the values of their Odds Stay Current w-day window (OSC-w). These ratios and values help determine the likelihood of borrowers staying current on their loans. We calculate the average of these values across the k nearest neighbors, considering various window periods (w) ranging from 1 day to 2^{14} days. This average represents the credit score and likelihood of staying current on future payments,



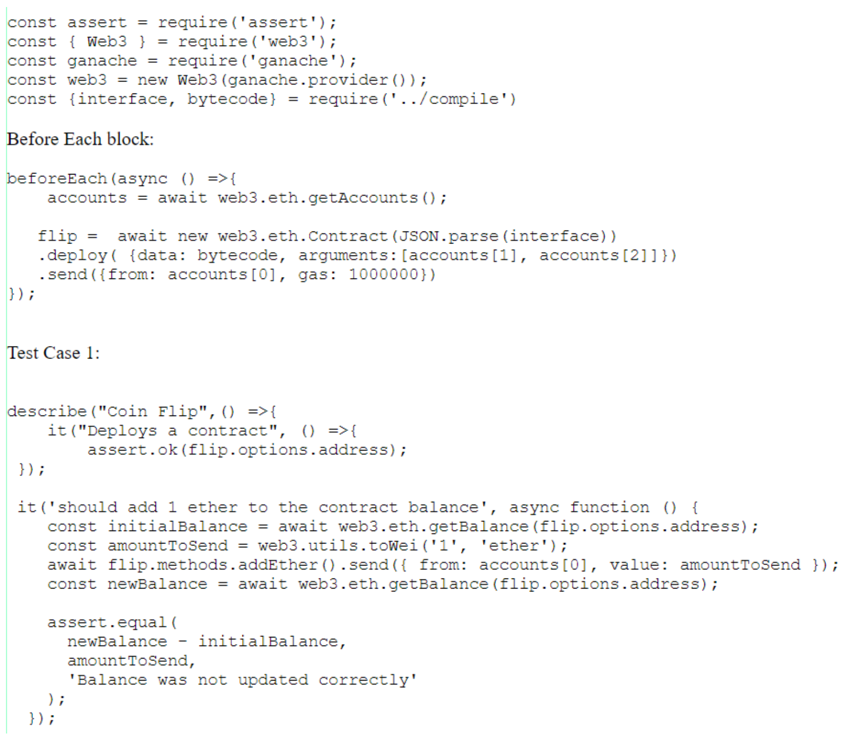
```
function getScore(address borrower) public view returns (uint) {
    ScorePoint[] memory points = _scorePoints[borrower];
    uint[] memory distances = new uint[](points.length);

    // If there are no score points, return default FICO score
    if (points.length == 0) {
        return _ficoScores[borrower].score * 1000;
    }

    for (uint i = 0; i < points.length; i++) {
        distances[i] = calculateDistance(points[i].score, _ficoScores[borrower].score);
    }

    // Sort distances array to find k-nearest neighbors
    sort(distances);

    // Get the k-nearest neighbors
    uint k = 5;
    uint totalScore = 0;
    for (uint i = 0; i < k; i++) {
        totalScore += points[i].score;
    }
    uint averageScore = totalScore / k;

    // Calculate UPR, CDBR, and OSCw for each nearest neighbor
    for (uint i = 0; i < k; i++) {
        uint upr = calculateUPR(points[i].borrower, points[i].amountOwed, points[i].amountShouldOwe);
        uint cdbr = calculateCDBR(points[i].borrower, points[i].currentOutstandingDebt, points[i].averageOutstandingDebt);
        uint oscw = calculateOSC(1, points[i].borrower);

        averageScore += adjustScore(upr, cdbr, oscw);
    }

    return averageScore;

}
```

Figure 22: CBSC: getScore method

V . Experiment and Results

In order to ensure the reliability and functionality of the lending system I developed, rigorous testing was essential. I employed a local blockchain environment for testing, leveraging various tools and frameworks to conduct comprehensive unit and integration tests.

For testing the smart contract, I employed the Mocha JavaScript test framework along with Ganache and Web3. Mocha simplifies asynchronous testing and provides detailed reporting capabilities. Ganache, on the other hand, furnishes a set of unlocked accounts for local testing purposes, eliminating the need for public or private keys. Web3 facilitates access to local or remote Ethereum nodes, and for testing, I connected to a local node provided by Ganache.



```
const assert = require('assert');
const { Web3 } = require('web3');
const ganache = require('ganache');
const web3 = new Web3(ganache.provider());
const {interface, bytecode} = require('../compile')
```

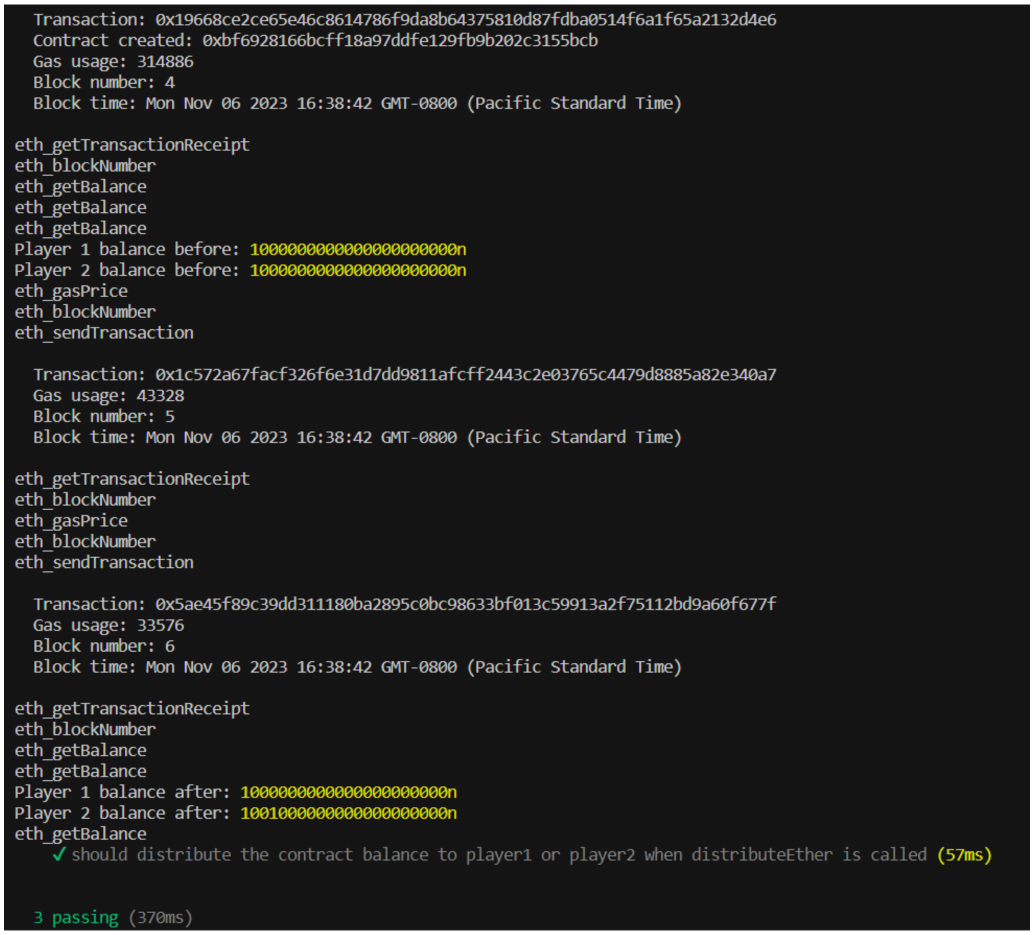
Before Each block:

```
beforeEach(async () =>{
    accounts = await web3.eth.getAccounts();

   flip =  await new web3.eth.Contract(JSON.parse(interface))
   .deploy( {data: bytecode, arguments:[accounts[1], accounts[2]]})
   .send({from: accounts[0], gas: 1000000})
});
```

Test Case 1:

```
describe("Coin Flip",() =>{
    it("Deploys a contract", () =>{
        assert.ok(flip.options.address);
 });

 it('should add 1 ether to the contract balance', async function () {
    const initialBalance = await web3.eth.getBalance(flip.options.address);
    const amountToSend = web3.utils.toWei('1', 'ether');
    await flip.methods.addEther().send({ from: accounts[0], value: amountToSend });
    const newBalance = await web3.eth.getBalance(flip.options.address);

    assert.equal(
      newBalance - initialBalance,
      amountToSend,
      'Balance was not updated correctly'
    );
  });
```

Figure 24: Local Blockchain: a sample test case



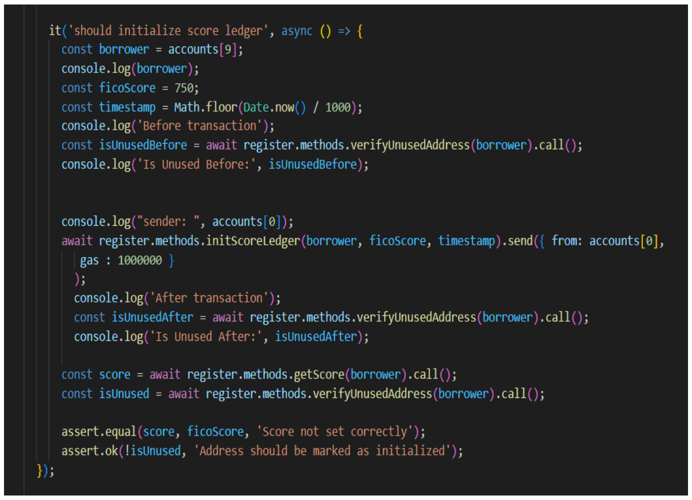
```
  Transaction: 0x19668ce2ce65e46c8614786f9da8b64375810d87fdba0514f6a1f65a2132d4e6
  Contract created: 0xbf6928166bcff18a97ddfe129fb9b202c3155bcb
  Gas usage: 314886
  Block number: 4
  Block time: Mon Nov 06 2023 16:38:42 GMT-0800 (Pacific Standard Time)

eth_getTransactionReceipt
eth_blockNumber
eth_getBalance
eth_getBalance
eth_getBalance
Player 1 balance before: 1000000000000000000000n
Player 2 balance before: 1000000000000000000000n
eth_gasPrice
eth_blockNumber
eth_sendTransaction

  Transaction: 0x1c572a67facf326f6e31d7dd9811afcff2443c2e03765c4479d8885a82e340a7
  Gas usage: 43328
  Block number: 5
  Block time: Mon Nov 06 2023 16:38:42 GMT-0800 (Pacific Standard Time)

eth_getTransactionReceipt
eth_blockNumber
eth_gasPrice
eth_blockNumber
eth_sendTransaction

  Transaction: 0x5ae45f89c39dd311180ba2895c0bc98633bf013c59913a2f75112bd9a60f677f
  Gas usage: 33576
  Block number: 6
  Block time: Mon Nov 06 2023 16:38:42 GMT-0800 (Pacific Standard Time)

eth_getTransactionReceipt
eth_blockNumber
eth_getBalance
eth_getBalance
Player 1 balance after: 1000000000000000000000n
Player 2 balance after: 1001000000000000000000n
eth_getBalance
    ✓ should distribute the contract balance to player1 or player2 when distributeEther is called (57ms)


  3 passing (370ms)
```

Figure 25: Local Blockchain: Test case output

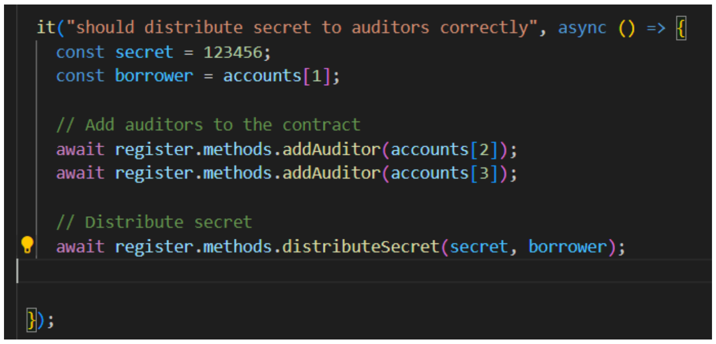
The registration phase of the lending system involves several key entities and processes to establish a borrower's identity and credit score. Functions implemented in the Registration smart contract:

- **initScoreLedger**: Initializes the credit score ledger for a borrower with their FICO score and timestamp. This function can only be called once per address.
- **verifyUnusedAddress**: Checks if a borrower's address has already been initialized with a credit score.
- **getScore**: Retrieves the credit score associated with a borrower's address.

- **addAuditor**: Adds an auditor to the system.
- **distributeSecret**: Implements a polynomial secret-sharing scheme inspired by Shamir's secret-sharing algorithm to distribute the borrower's SSN among auditors. This function ensures that no single auditor has access to the complete SSN by dividing it into shares.
- **reconstructSecret**: Enables the reconstruction of the original secret from the distributed shares using principles from Shamir's algorithm. This function preserves data privacy and confidentiality by reconstructing the SSN only when necessary, such as in the case of a loan default.



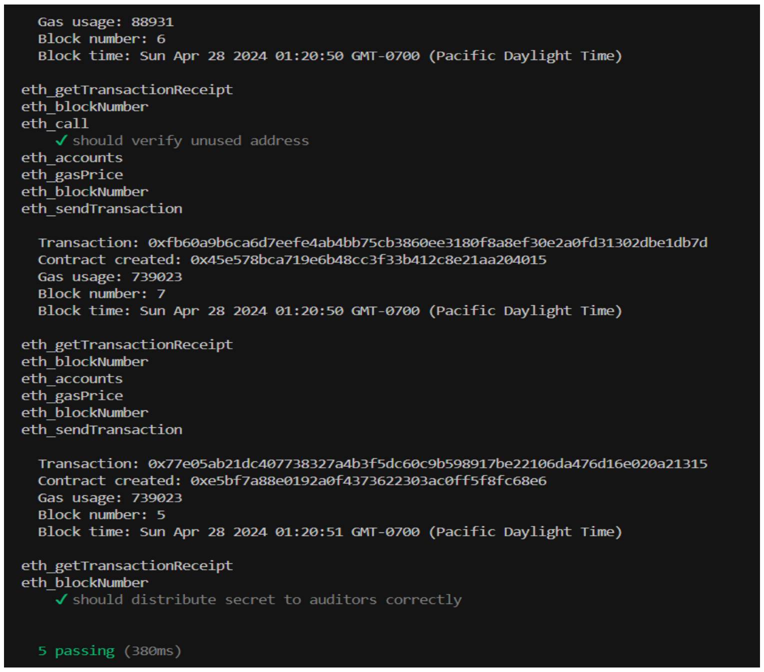
```
it('should initialize score ledger', async () => {
  const borrower = accounts[9];
  console.log(borrower);
  const ficoScore = 750;
  const timestamp = Math.floor(Date.now() / 1000);
  console.log('Before transaction');
  const isUnusedBefore = await register.methods.verifyUnusedAddress(borrower).call();
  console.log('Is Unused Before:', isUnusedBefore);


  console.log("sender: ", accounts[0]);
  await register.methods.initScoreLedger(borrower, ficoScore, timestamp).send({ from: accounts[0],
    gas : 1000000 }
    );
    console.log('After transaction');
    const isUnusedAfter = await register.methods.verifyUnusedAddress(borrower).call();
    console.log('Is Unused After:', isUnusedAfter);

  const score = await register.methods.getScore(borrower).call();
  const isUnused = await register.methods.verifyUnusedAddress(borrower).call();

  assert.equal(score, ficoScore, 'Score not set correctly');
  assert.ok(!isUnused, 'Address should be marked as initialized');
});
```



```
it("should distribute secret to auditors correctly", async () => {
  const secret = 123456;
  const borrower = accounts[1];

  // Add auditors to the contract
  await register.methods.addAuditor(accounts[2]);
  await register.methods.addAuditor(accounts[3]);

  // Distribute secret
  await register.methods.distributeSecret(secret, borrower);


});
```

Figure 26: Borrower Registration: Test cases

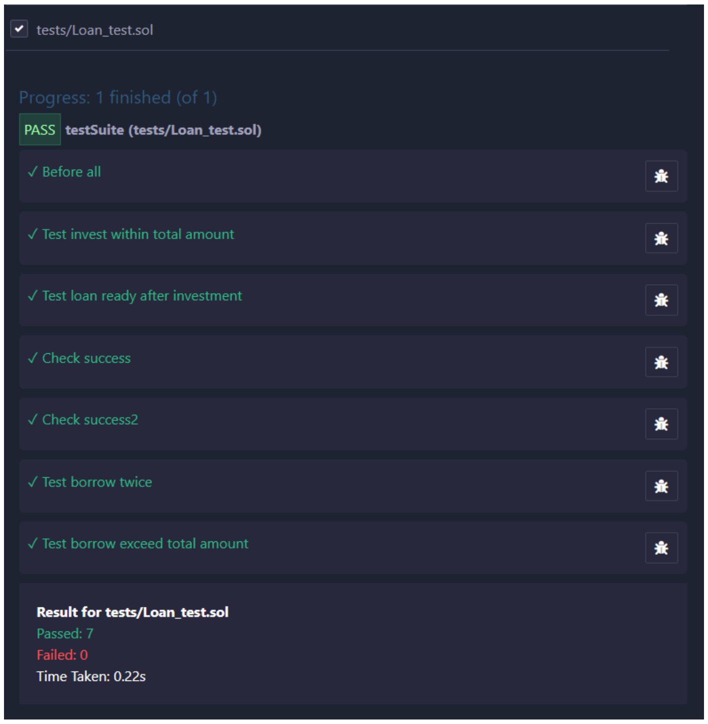
Unit tests were conducted to validate the registration phase functionalities. They include verifying the successful deployment of the contract and ensuring the proper initialization of account balances. These tests are vital for securely managing borrower identities and credit scores within the system. Additionally, a test case was developed to validate the distribution of secrets, implementing a polynomial secret-sharing scheme inspired by Shamir's algorithm.



```
  Gas usage: 88931
  Block number: 6
  Block time: Sun Apr 28 2024 01:20:50 GMT-0700 (Pacific Daylight Time)

eth_getTransactionReceipt
eth_blockNumber
eth_call
    ✓ should verify unused address
eth_accounts
eth_gasPrice
eth_blockNumber
eth_sendTransaction

  Transaction: 0xfb60a9b6ca6d7eefe4ab4bb75cb3860ee3180f8a8ef30e2a0fd31302dbe1db7d
  Contract created: 0x45e578bca719e6b48cc3f33b412c8e21aa204015
  Gas usage: 739023
  Block number: 7
  Block time: Sun Apr 28 2024 01:20:50 GMT-0700 (Pacific Daylight Time)

eth_getTransactionReceipt
eth_blockNumber
eth_accounts
eth_gasPrice
eth_blockNumber
eth_sendTransaction

  Transaction: 0x77e05ab21dc407738327a4b3f5dc60c9b598917be22106da476d16e020a21315
  Contract created: 0xe5bf7a88e0192a0f4373622303ac0ff5f8fc68e6
  Gas usage: 739023
  Block number: 5
  Block time: Sun Apr 28 2024 01:20:51 GMT-0700 (Pacific Daylight Time)

eth_getTransactionReceipt
eth_blockNumber
    ✓ should distribute secret to auditors correctly


  5 passing (380ms)
```

Figure 27: Borrower Registration: test cases

The Loan Smart Contract serves as a vital intermediary, linking lenders and borrowers within the system. Implemented functionalities include *invest, borrow,* and *interest* calculation methods. The constructor initializes essential variables such as the total loan amount, interest rate, and minimum credit score requirement. Testing the invest function involves scenarios to verify correct fund allocation and timestamp initialization. Similarly, the borrow function undergoes tests to ensure proper loan requests and credit score validation. Additionally, the *calculateInterest* function is rigorously tested to validate accurate interest computation based on specified parameters. Lastly, the *withdraw* function permits investors to withdraw funds from the contract, adding a crucial layer of functionality to the lending system.



```
tests/Loan_test.sol

Progress: 1 finished (of 1)
PASS testSuite (tests/Loan_test.sol)
✓ Before all
✓ Test invest within total amount
✓ Test loan ready after investment
✓ Check success
✓ Check success2
✓ Test borrow twice
✓ Test borrow exceed total amount

Result for tests/Loan_test.sol
Passed: 7
Failed: 0
Time Taken: 0.22s
```

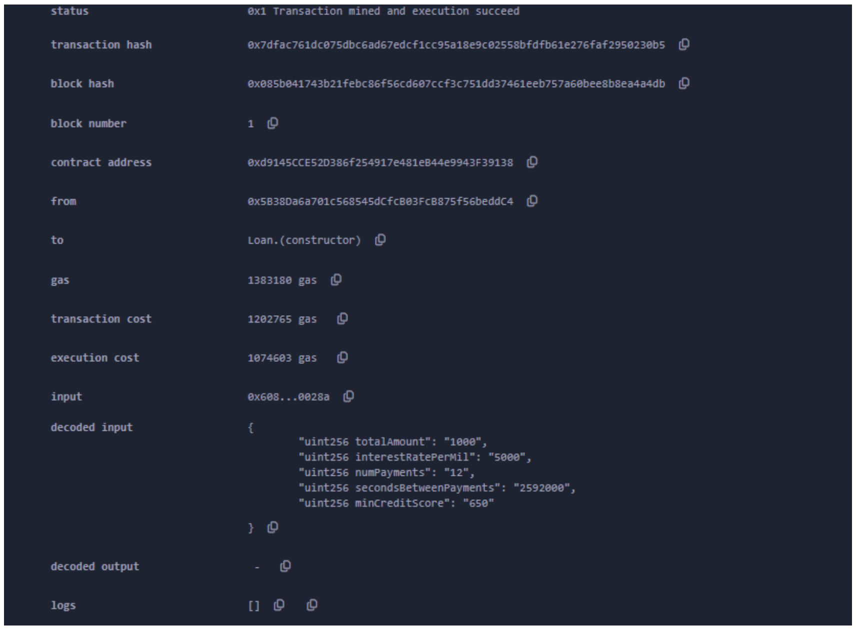
Figure 28: Loan Smart Contract: Test Suite

Transaction fees associated with deploying the Smart Contracts and implementing some functions are as follows:

Deploying contract	1383180 gas
Borrow() (involves a call to isReady())	119606 gas
Invest()	107126 gas
isReady() (involves a call to getScore())	38552 gas
_amountInvested()	2495 gas
initScoreLedger()	211681 gas

get$$$()	27965 gas
getScore()	29396 gas

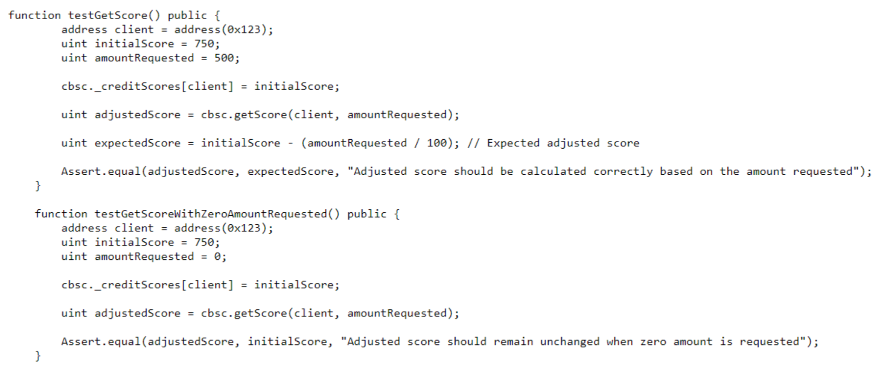
Table 2. Transaction costs measured in gas



```
status                 0x1 Transaction mined and execution succeed
transaction hash       0x7dfac761dc075dbc6ad67edcf1cc95a18e9c02558bfdfb61e276faf2950230b5
block hash             0x085b041743b21febc86f56cd607ccf3c751dd37461eeb757a60bee8b8ea4a4db
block number           1
contract address       0xd9145CCE52D386f254917e481eB44e9943F39138
from                   0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                     Loan.(constructor)
gas                    1383180 gas
transaction cost       1202765 gas
execution cost         1074603 gas
input                  0x608...0028a
decoded input          {
                               "uint256 totalAmount": "1000",
                               "uint256 interestRatePerMil": "5000",
                               "uint256 numPayments": "12",
                               "uint256 secondsBetweenPayments": "2592000",
                               "uint256 minCreditScore": "650"
                       }
decoded output          -
logs                   []
```

Figure 29: Transaction costs when deploying a Smart Contract

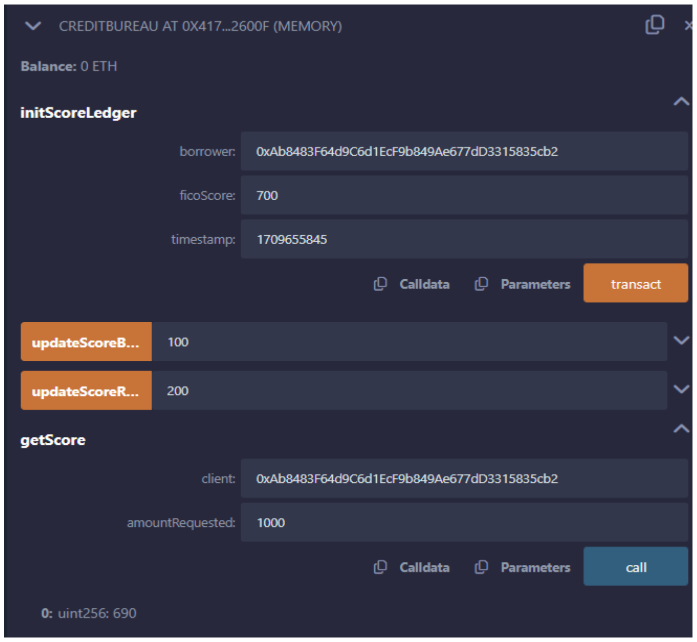
In conclusion, this report describes the successful implementation and testing of essential functionalities within the lending system. Leveraging smart contracts and blockchain technology, a robust framework for managing borrower identities, credit scores, and loan transactions has been established. Rigorous testing, including unit and integration tests, ensures the reliability and correctness of the system. As mentioned before, the Credit Bureau Smart Contract (CBSC) is a pivotal component, in managing the association between Ethereum addresses and credit scoring data. This includes the hashed identity, timestamp of linkage, and credit score position. Implemented functionalities, including updateScoreBorrow, updateScoreRepayment, and getScore methods, to ensure seamless borrower-lender connections and credit score management. Emphasis is placed on rigorous testing, with scenarios covering various use cases. For instance, the getScore function is meticulously tested to ensure accurate adjustments to the client's credit score based on the requested amount.



```
function testGetScore() public {
        address client = address(0x123);
        uint initialScore = 750;
        uint amountRequested = 500;

        cbsc._creditScores[client] = initialScore;

        uint adjustedScore = cbsc.getScore(client, amountRequested);

        uint expectedScore = initialScore - (amountRequested / 100); // Expected adjusted score

        Assert.equal(adjustedScore, expectedScore, "Adjusted score should be calculated correctly based on the amount requested");
    }

    function testGetScoreWithZeroAmountRequested() public {
        address client = address(0x123);
        uint initialScore = 750;
        uint amountRequested = 0;

        cbsc._creditScores[client] = initialScore;

        uint adjustedScore = cbsc.getScore(client, amountRequested);

        Assert.equal(adjustedScore, initialScore, "Adjusted score should remain unchanged when zero amount is requested");
    }
```

Figure 29: Credit Bureau Smart Contract: Testing getScore()

Additionally, basic versions of updateScoreBorrow and updateScoreRepayment are implemented and tested to guarantee the proper functioning of credit score updates during loan acquisition and repayment.



CREDITBUREAU AT 0X417...2600F (MEMORY)

Balance: 0 ETH

initScoreLedger

borrower: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

ficoScore: 700

timestamp: 1709655845

100

200

getScore

client: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

amountRequested: 1000

0: uint256: 690

Figure 30: Deploying Credit Bureau Smart Contrat

VI. Conclusion

This lending system implements the Autonomous Lending Organization on Ethereum with Credit Scoring as outlined by Thomas Austin, et. al. in [1]. The purpose of the project was to both understand the system's intricacies and the challenges of a Decentralized finance (DeFi) product that is built on the Ethereum platform. Upon establishing the foundation for programming smart contracts for the Ethereum blockchain platform using Solidity, In the first phase, I studied how Ethereum provides a system for electronic transactions without relying on a trusted third party. To gain a better understanding of Blockchain, I studied Merkle trees and Byzantine agreement for Bitcoin. I then set up a local blockchain, that facilitated the testing and deployment of the smart contract, using Web3.js JavaScript library and Ganache that simulates the Ethereum network. Phase 2 began with the implementation of the borrower registration phase, the borrower discloses their real-world identity, specifically their social security number (SSN), to the credit bureau. This contract aims to provide a secure and decentralized method for initializing borrowers and managing credit scores, ensuring confidentiality through encryption, and sharing the information with designated auditors. Next was the implementation of the Loan Smart Contract. A loan smart contract object is created by calling the *creatLoan()* method in the Credit Bureau Smart Contract. To engage with a Loan smart contract, the borrower or lender initiates the *borrow(uint amount)* or *invest* method, respectively.

Credit Bureau Smart Contract (CBSC) plays a pivotal role within the system by maintaining a mapping that correlates Ethereum addresses with pertinent credit scoring data. This supplementary information encompasses the credit score associated with the hashed identity, along with a timestamp denoting the inception of this linkage and a position within our credit scoring spectrum. Acting as a vital intermediary, the CBSC fosters connections between lenders and borrowers, oversees loan transactions and repayments, and orchestrates updates to the borrower's credit score. Noteworthy is the fact that exclusive authority to trigger designated methods of this smart contract is vested in the notary, ensuring governance and oversight over critical operations within the system.

The system is set to empower borrowers to initiate loan requests and facilitates seamless interaction between borrowers and lenders, offering lenders the capability to assess and engage with loan requests effectively. It ensures seamless connectivity and communication among all stakeholders involved in the lending process. In addition, this system permits the disclosure of borrowers' identities in the event of loan default, while simultaneously safeguarding the anonymity of borrowers who adhere to loan repayment terms.

A comprehensive suite of unit test cases has been devised and executed to rigorously evaluate the functionality and integrity of the developed codebase within this project.

VII. Future Work

In future development, the project aims to extend the system by implementing the generalized Shamir's secret sharing algorithm to enhance borrower identity protection, while maintaining auditor access in case of default. Another approach would be to take the implementation of the secret sharing algorithm off-chain, to enhance the security and privacy of the borrower. There are various independently audited, zero-dependency TypeScript implementations of Shamir's Secret Sharing algorithm.

7.1 Credit Scoring Model

Further, refining the nearest neighbor model by including additional metrics such as the Current Payment Burden Ratio and Repayment Age Ratio (RAR) to provide deeper insights into borrower behavior and credit history. Next, introducing the Average Number of Credit Lines (ANCL) metric to assess borrower credit diversity and history. Exploring the integration of an off-chain oracle to reduce computation costs for the nearest neighbor model, while ensuring security standards are upheld.

7.2 Implementation

In the upcoming phase, the React framework will be leveraged to construct the user interface (UI) for the lending system based on the paper by Thomas Austin et. al. [1]. By harnessing React's component-based architecture, we can create modular and reusable UI elements, streamlining the development process and fostering code maintainability. Additionally, React's virtual DOM ensures optimal performance, enabling seamless rendering of UI components and enhancing the overall user experience. Through the utilization of React, we aim to craft a sophisticated and intuitive UI that empowers users to interact with our system effortlessly while delivering a visually appealing and engaging interface for lenders and borrowers.

REFERENCES

[1] T. H. Austin, K. Potika and C. Pollett, 'Autonomous Lending Organization on Ethereum with Credit Scoring,' 2023 Silicon Valley Cybersecurity Conference (SVCC), San Jose, CA, USA, 2023.

[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008 [Online]

[3] V. Buterin, Ethereum white paper, 2013. Available online: https://github.com/ethereum/wiki/wiki/White-Paper

[4] C. Busayatananphon and E. Boonchieng, 'Financial Technology DeFi Protocol: A Review,' 2022 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT and NCON), Chiang Rai, Thailand, 2022.

[5] L. Andolfo, L. Coppolino, S. DAntonio, G. Mazzeo, L. Romano,M. Ficke, A. Hollum, and D. Vaydia, 'Privacy-preserving credit scoring via functional encryption,' in International Conference on Computational Science and Its Applications, 2021

[6] 'The solidity contract-oriented programming language.' https://github.com/ethereum/, accessed November 2020

[7] C. Lin, M. Luo, X. Huang, K.-K. R. Choo, and D. He, "An efficient privacy-preserving credit score system based on noninteractive zero knowledge proof," IEEE systems journal, 2021.

[8] Ahmed, Mamun & Reno, Saha & Akter, Salma & Chowdhury, A., " Decentralized Finance

and Crypto Banking System Using Ethereum-based Blockchain Technology". 2. 33-51 (2022).

[9] Yang, S. and Cui, W, "An Evaluation System for DeFi Lending Protocols", 2023 42nd Chinese Control Conference (CCC)

[10] Huber, M., & Treytl, V. (2022), " Risks in DeFi-Lending Protocols - An Exploratory Categorization and Analysis of Interest Rate Differences" Communications in Computer and Information Science, 1633 CCIS, 258–269.

[11] Legowo, N., Hawari, N., Karlina, T., Tanuwijaya, E., & Mahendra, K., " Design Smart Contract Based on Blockchain for Peer-To-Peer Lending Platform", 10th International Conference on ICT for Smart Society, ICISS 2023

[12]Ta, M. T., & Do, T. Q. (2024). A study on gas cost of Ethereum smart contracts and performance of blockchain on simulation tool. Peer-to-Peer Networking and Applications, 17(1), 200–212. https://doi-org.libaccess.sjlibrary.org/10.1007/s12083-023-01598-3

[13] Chaleenutthawut, Yatipa, et al. "Loan Portfolio Dataset From MakerDAO Blockchain Project." IEEE Access, vol. 12, 2024, pp. 24843–54, https://doi.org/10.1109/ACCESS.2024.3363225.

[14] Qin, Kaihua, et al. "An Empirical Study of DeFi Liquidations." Proceedings of the 21st ACM Internet Measurement Conference, ACM, 2021, pp. 61–350, https://doi.org/10.1145/3487552.3487811.