Credit score-based lending system on the Ethereum Platform

A Project Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Class

CS 297

By

Mayuri Shimpi

December 2023

I. ABSTRACT

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 project focuses on implementing 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 current semester's objectives include establishing a robust understanding of cryptocurrencies and the Ethereum platform and implementing pivotal components of the ALOE system, as presented by Austin, Potika, and Pollett in 2023. Specifically, the project aims to build the borrower registration and borrowing processes while incorporating 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.

TABLE OF CONTENTS

I. INTRODUCTION

The advent of modern cryptocurrencies like Ethereum has revolutionized the financial landscape by introducing the concept of smart contracts. These self-executing agreements operate on a decentralized blockchain network, eliminating the need for intermediaries like banks. In the realm of lending, smart contracts enable direct peer-to-peer transactions, empowering individuals to borrow and lend funds without the involvement of traditional financial institutions.

The primary objective of this project is to deploy the Autonomous Lending system on the Ethereum Platform (ALOE), as outlined by T. Austin et.al in [1]. The goal is to achieve a seamless integration of conventional credit scoring methodologies to assess the risk of default for potential borrowers. This system tackles the crucial aspects of identity verification, creditworthiness assessment, and borrower-lender matching, paving the way for a new era of accessible and inclusive financial services.

The Credit score-based lending system facilitates peer-to-peer, unsecured loans on the blockchain, eliminating the need for intermediaries like banks while striving for maximum transparency by conducting as much as possible on the blockchain, where the computations in contracts can be fully examined. This includes the credit-scoring mechanism, ensuring that all calculations are open to public scrutiny. This system is specifically developed for the Ethereum blockchain, utilizing its virtual machine and Solidity as the high-level language. Ethereum's robust infrastructure and smart contract capabilities provide a solid foundation for the system.

The report is organized into several key sections, each addressing specific deliverables and

aspects of the project. It begins with Deliverable I, focusing on the creation of a Coin-Flip smart contract. Following this, Deliverable II is explored, detailing the setup of a local blockchain. Deliverable III is then covered, which involves the implementation of the Borrower Registration process. The report proceeds to Deliverable IV, where existing architectures are researched, and the system design is outlined. The findings and insights from the project are summarized in the Conclusion, followed by a list of References.

II. DELIVERABLE 1: CREATE A COIN-FLIP SMART CONTRACT

Smart contracts allow developers to build a wide variety of decentralized apps and tokens. These applications span across a range of sectors, from innovative financial tools to logistics and gaming experiences. Similar to other cryptocurrency transactions, smart contracts are stored on a blockchain. Once a smart contract application is added to the blockchain, reversing or altering it is typically not possible.

Web3.js is a JavaScript library that provides an interface for interacting with the Ethereum blockchain. It enables developers to build applications that can interact with smart contracts, query blockchain data, and send transactions.

Setting up the development environment for Ethereum smart contract development is a practical step that enhances my technical skills. Through this, I gain hands-on experience with the tools and platforms essential for actual project implementation.

1. Go to the Remix online IDE.

Remix is an Ethereum-focused IDE: an online platform to develop and deploy smart contracts. To build a smart contract, I created a new file called _coinFlip.sol.

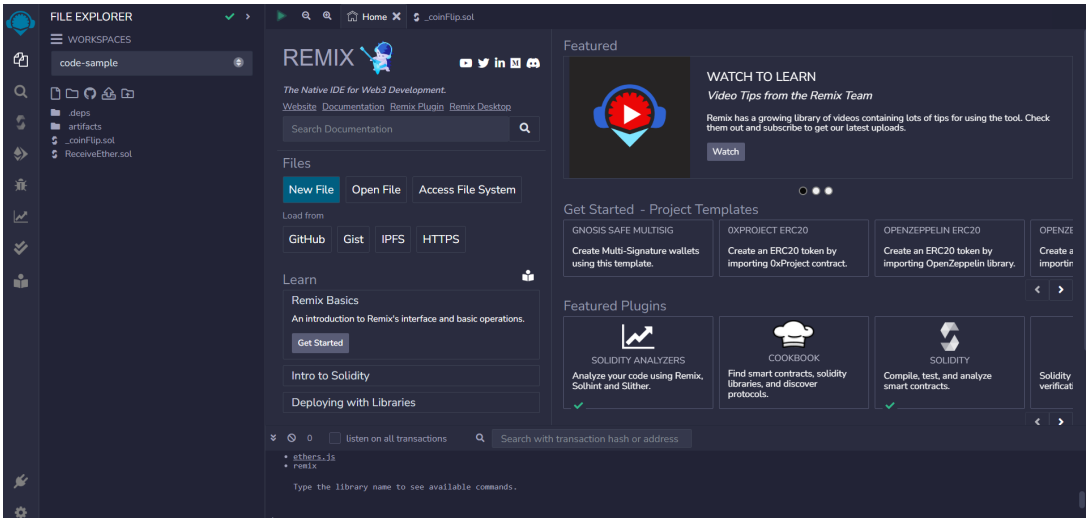

Figure 1. The RemixIDE

2. Writing a smart contract: A smart contract is a self-executing program that automates the actions required in an agreement or contract. Once completed, the transactions are trackable and irreversible.

My smart contract code:

```
// 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) {
        sender = msg.sender;
        player1 = _player1;
        player2 = _player2;
        balance = 0;
    }

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

```
    function distribute() external {
        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;
    }
}
```

The first line pragma solidity >=0.7.0 <0.9.0 specifies that the source code is for a Solidity version greater than 0.7.0 and lesser than 0.9.0.

Pragmas are common instructions for compilers about how to treat the source code (e.g., pragma once).

This is a simple Ethereum smart contract that facilitates a coin-flip game between two players. This contract essentially creates a simple game where players can add Ether to a common pool, and the sender can trigger a distribution based on the outcome of a coin flip. The distribution is biased based on the parity of the last digit of the current block's timestamp.

3. Compile Smart Contract:

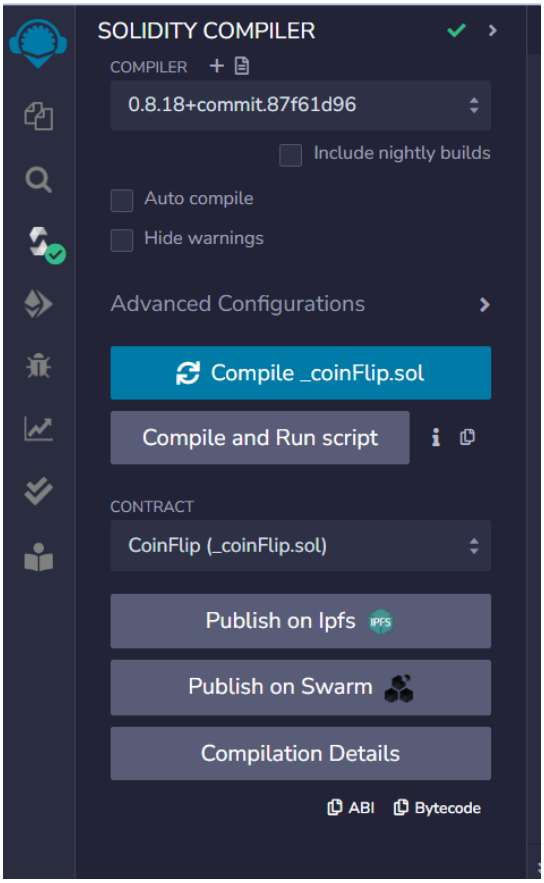

Figure 2. Compile the contract

After successful compilation, it will show a green tick mark on the Compiler tab button.

4. Deploying the contract: The contract was deployed on the Remix VM for testing purposes. For this coin flip contract, there are 2 players and a sender. When deploying the contract, an address is selected for the Sender, and the addresses for players 1 and 2. The provided accounts are just test IDs and are only meant for testing purposes.

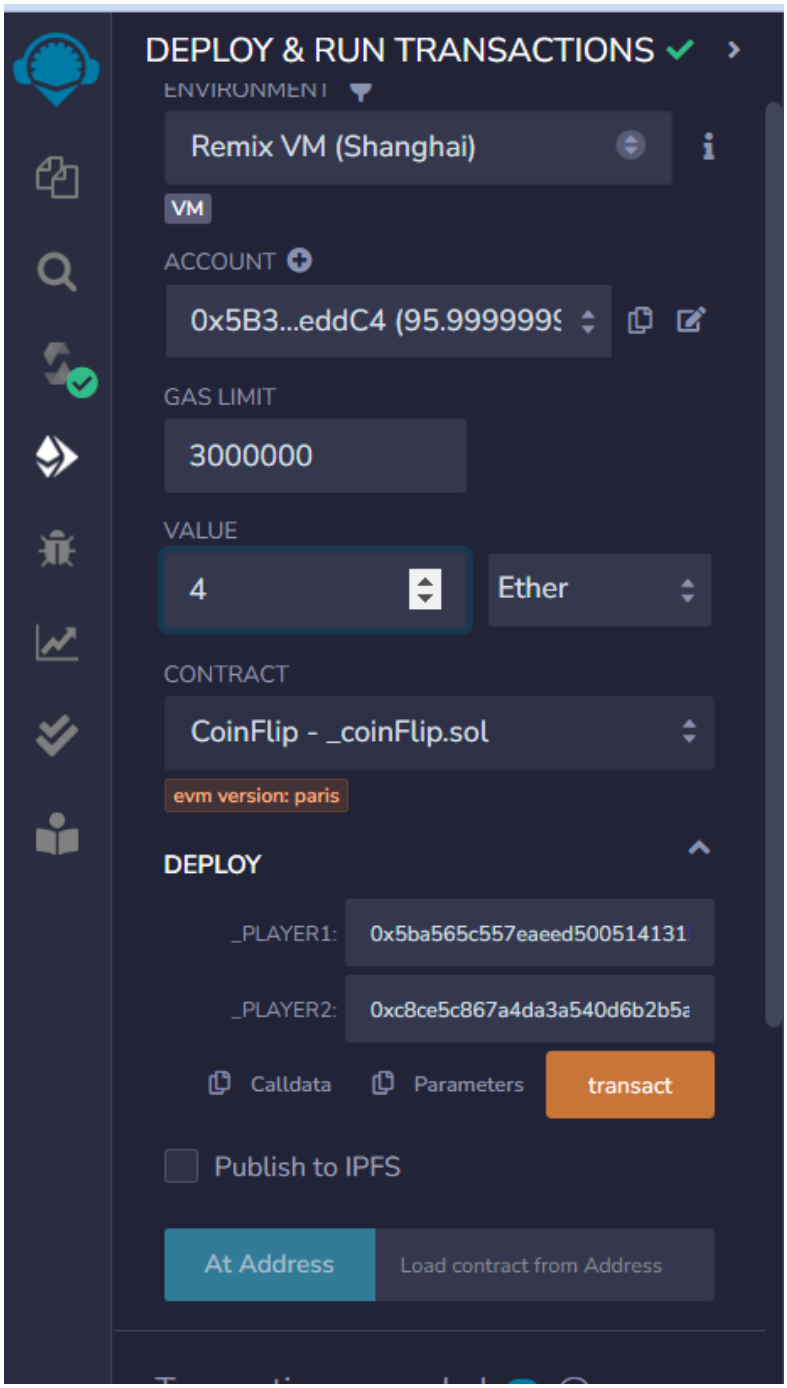

Figure 3. Deploy the contract for testing

5. Test the Coin Flip game:

To test this contract, it is necessary to transfer test Ether from the Sender's account to the smart contract's balance. To accomplish this, I have chosen to allocate 4 Ethers and

executed the "addEther" function to append this balance (refer to Figure 4). Upon inspecting the "balance," the current balance is indicated as 4 ETH. Subsequently, the "distributeEther" function is employed to randomly choose either 0 or 1, redistributing the balance to either Player 1 or Player 2, depending on the outcome. The final illustration demonstrates the reduction in the sender's balance and the addition of 4 ETH to Player 1's account.

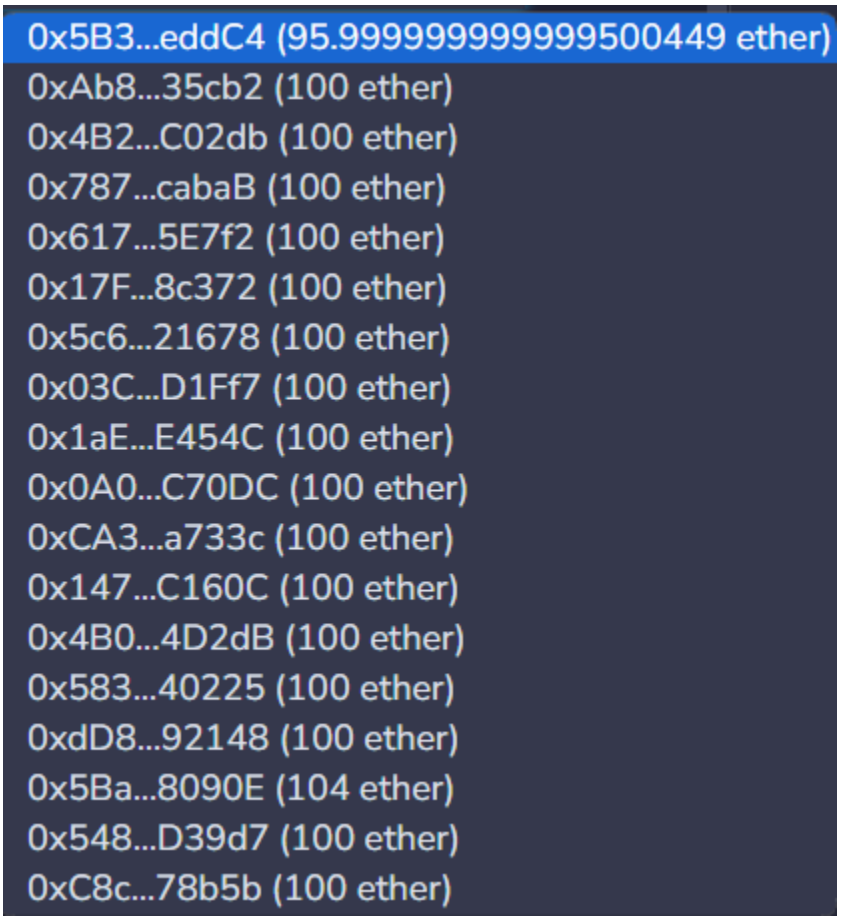

Figure 4. Transactions and Ether balance indicate that Player 1 won.

Thus, the smart contract was successfully written, compiled, deployed and tested using the Remix VM.

III. DELIVERABLE 2: SETTING UP A LOCAL BLOCKCHAIN

Setting up a local blockchain environment is valuable for various reasons, especially during the development and testing phases of blockchain applications. To facilitate the testing and deployment of the smart contracts, I used the Web3.js JavaScript library. This library serves as an interface for interacting with the Ethereum blockchain, empowering developers to construct applications that seamlessly engage with smart contracts, retrieve blockchain data, and execute transactions.

The "_coinFlip.sol" file was compiled to extract essential properties, namely the Application Binary Interface (ABI) and the bytecode. The bytecode is deployed onto the blockchain to activate the smart contract, while the ABI acts as a human-friendly JavaScript layer mapping the bytecode.

To initiate this process, I installed solc, a Solidity compiler, by executing the command "npm install --save solc" in the terminal at the project's root. Subsequently, I created a file named "compile.js" at the project's root, utilizing Node modules (path, fs) and the Solidity compiler (solc). Due to Node's behavior, which attempts to execute files as JavaScript, direct requiring of the .sol file is not feasible. Instead, the source variable captures the raw source code of "_coinFlip.sol," employing UTF-8 encoding.

Executing the compile() function on the source code, specifying the number of contracts to compile (in this case, one), yields the necessary output. Upon compiling the code using the command "node compile.js," the following output is generated:

Figure 5: The output upon compiling the smart contract with solc, a Solidity compiler.

The bytecode is the code that will be deployed onto the blockchain and executed in the Ethereum Virtual Machine (EVM), which is not human-readable.

Let us take a closer look at the interface:

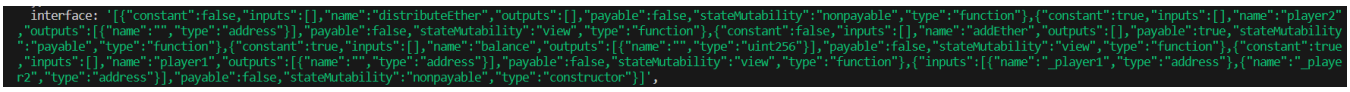

Figure 6: The Application Binary Interface (ABI)

Testing the smart contract: Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

Ganache and Web3 libraries were used in addition to Mocha. Ganache provides a set of unlocked accounts we will use in our local testing environment. In this case, unlocked means that there is no need to use a public or private key to access them. Web3 provides libraries that allow access to a local or remote Ethereum node. For testing purposes, the connection was established to a local node, which was facilitated by Ganache.

The following commands are used to install the required libraries:

```
npm install --save mocha ganache-cli web3@4.1.2
```

Next, create a test folder with the file named _coinFlip.test.js.

First, require the Mocha assertion library, Ganache, and Web3. Notice that we capitalize Web3. It is a constructor which will be used to create an instance of it in our test file.

The _coinFlip.test.js file:

```
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'
    );
  });
```

Test Case 2:

```
  it('should distribute the contract balance to player1 or player2 when
distributeEther is called', async () => {
    const initialBalance = await web3.eth.getBalance(flip.options.address);
    const amountToSend = web3.utils.toWei('1', 'ether');
    const player1BalanceBefore = await web3.eth.getBalance(accounts[1]);
    const player2BalanceBefore = await web3.eth.getBalance(accounts[2]);
```

```
    console.log("Player 1 balance before:", player1BalanceBefore);
    console.log("Player 2 balance before:", player2BalanceBefore)
    // Add 1 ETH to the contract balance
    await flip.methods.addEther().send({ from: accounts[0], value: amountToSend
});

    // Call distributeEther
    await flip.methods.distributeEther().send({ from: accounts[0] });
    const player1BalanceAfter = await web3.eth.getBalance(accounts[1]);
    const player2BalanceAfter = await web3.eth.getBalance(accounts[2]);
    console.log("Player 1 balance after:", player1BalanceAfter);
    console.log("Player 2 balance after:", player2BalanceAfter)
    const newBalance = await web3.eth.getBalance(flip.options.address);
    assert.equal(newBalance, 0, 'Balance was not reset to 0');
  });
});
```

Test case outputs:

```
  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: 100000000000000000000n
Player 2 balance before: 100000000000000000000n
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: 100000000000000000000n
Player 2 balance after: 100100000000000000000n
eth_getBalance
    ✓ should distribute the contract balance to player1 or player2 when distributeEther is called (57ms)


  3 passing (370ms)
```

Figure 7. The output of the test cases shows that all 3 passed successfully.

IV. DELIVERABLE 3: CREDIT BUREAU SMART CONTRACT: REGISTRATION PHASE

During the registration phase, the client discloses their real-world identity, specifically their social security number (SSN), to the credit bureau. The registration process involves various entities:

Borrower: This is an actual individual with a unique SSN and an associated Ethereum address.

Notary: Serving as a trusted real-world entity, the notary validates the user's identity, credit score, and Ethereum address. Following validation, the notary divides the user's identity into secret shares for multiple auditors. The notary then initiates the credit bureau smart contract, establishing an initial credit score for the borrower. This score comprises the saved real-world score and a point in a credit space, the details of which will be explained in the next section. It is assumed that the notary does not retain the association between the borrower and their Ethereum address, except for the secret shares sent to the auditors.

Auditors: Responsible for storing shares of the borrower's real-world identity, auditors are trusted

entities. It is expected that they do not disclose their share of the secret unless specified by the protocol. For simplicity, the assumption is made that the set of auditors is small, fixed, and publicly known. These assumptions could potentially be relaxed with additional infrastructure.

Registration Smart Contract (RSC): The RSC maintains a mapping between Ethereum addresses and associated credit scoring information. This additional information includes the credit score linked to the hashed identity, a timestamp indicating when this association was established, and a point within the credit scoring space. The RSC is tasked with connecting lenders and borrowers, tracking loans and repayments, and updating the borrower's credit score. Importantly, only the notary has the authority to invoke specific methods of this smart contract.

Here is the Registration smart contract:

The `verifyBorrower` function verifies the credit scores using a decentralized and secure approach and divides the user's identity into secret shares for multiple auditors. Here's a breakdown of its key features:

```
 function verifyBorrower(address borrower, string memory ficoScore,string memory
ssn, uint timestamp) public returns (bool){
        require(!_scoreInitialized[borrower], "Credit score already initialized
       for this address");
         _creditScores[borrower] = ficoScore;
         _scoreTimestamps[borrower] = timestamp;
         _scoreInitialized[borrower] = true;

        // Convert the concatenated string to bytes
        bytes32 byteArray = bytes32(abi.encodePacked(ssn, ficoScore));
        bytes32 randomData = keccak256(abi.encodePacked(block.timestamp,
        blockhash(block.number - 1)));

        bytes32 cipher = xorBytesWithKey(byteArray, randomData);
        bytes32 original = xorBytesWithKey(cipher, randomData);

        // divides the user's identity into secret shares for multiple auditors
        _auditors[0].share = cipher;
        _auditors[1].share = randomData;
```

```
        if(original == byteArray)
            return true;
        else
            return false;

    }
```

This function verifies the borrower is unique and sets up the scoringfor a given borrower_account. The score ledger can be thought of as a ledger recording real-world FICO scores, borrowed amounts in the crypto setting, loan repayments, etc. This method verifies that the Credit score is not already initialized for this address to ensure that the address has not been used previously.

Then, divides the user's identity into secret shares for multiple auditors.

Unit test cases were written to test the functionality of this registration phase. For example:

```
  it('should initialize the credit score ledger for a borrower', async () => {
    const borrower = accounts[1];
    const ficoScore = '750';
    const ssn = '123456789';
    const timestamp = Math.floor(Date.now() / 1000); // Current timestamp in
seconds

    const result = await notary.verifyBorrower(borrower, ficoScore, ssn,
timestamp);

    assert.isTrue(result, 'Credit score should be initialized successfully');

    const isInitialized = await notary._scoreInitialized(borrower);
    const creditScore = await notary._creditScores(borrower);
    const scoreTimestamp = await notary._scoreTimestamps(borrower);
    const auditorShare1 = await notary._auditors(0);
    const auditorShare2 = await notary._auditors(1);

    assert.isTrue(isInitialized, 'Credit score should be marked as
initialized');
    assert.equal(creditScore, ficoScore, 'Credit score should match the
provided value');
    assert.equal(scoreTimestamp, timestamp, 'Timestamp should match the
provided value');

    // Perform XOR decryption and check the original value
    const original = await notary.xorBytesWithKey(auditorShare1.share,
```

```
auditorShare2.share);
    const expectedOriginal = web3.utils.soliditySha3(ssn, ficoScore);
    assert.equal(original, expectedOriginal, 'XOR decryption should match the
original value');
  });
```

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. The use of XOR encryption and the division of secret shares enhance the security of the credit score data.

V. DELIVERABLE 4: DEVELOPING THE LOAN SMART CONTRACT

This deliverable centers around the Loan Smart Contract, serving as an intermediary to connect lenders and borrowers. The Credit Bureau Smart Contract creates a new loan that borrowers and lenders can associate with. In our system, lenders can invest funds, and borrowers can specify the funds they require. After that, the interest rate is calculated based on the terms set by the EOA that initializes the Loan Smart Contract. The system allows borrowers to make payments and to track their credit scores, we update the CreditBureau smart contract with the capability of updating the credit score of the borrower. The system also allows lenders to make withdrawals.

Initiating this method is exclusive to the Externally Owned Account (EOA), referred to as the loan initiator. The loan initiator, responsible for providing ether for the operation, can claim fees from borrower interest and any additional interest resulting from early lender fund claims. These fees incentivize loan creators to structure loans with parameters conducive to attracting ample investors

and borrowers, ensuring successful funding and timely repayments.

In this deliverable, I have implemented the invest, borrow, and calculate interest methods as described below:

Constructor: This constructor defines and initializes variables for the Loan smart contract. The contract tracks the total loan amount, interest rate, number of payments, time intervals between payments, and a minimum credit score requirement. The variables _amountInvested and _amountBorrowed are initialized to 0 and are used for tracking investment and borrowing activity within the contract.

The **invest()** function: This function withdraws the amount specified by msg.value from the lender (msg.sender) and adds the balance to the loan smart contract. If sufficient funds are received, the timestamp of this transaction is utilized as the starting time for the loan. This timestamp is then employed to calculate the accrued interest.

The **borrow(uint amount)** function: This function is utilized by the borrower (msg.sender) to request a loan. The borrower's credit score and the requested amount must meet the loan requirements. This method involves a call to "getScore()", which will be elaborated on below and may incur gas costs. Similarly, the findLoan operation mentioned earlier incurs gas costs for computation and may have already computed the same getScore. To prevent costly recomputations, these calls may be cached. I haven't called the getScore() function yet; however, I plan to do so in the future. The isReady() function returns whether or not the loan has been funded.

The **calculateInterest(uint owed, uint numPayments)** function: This function, named calculateInterest, is designed to compute the accrued interest on a loan based on the provided

parameters.

To finalize the development of this smart contract, I will proceed to implement the remaining functions, including get$$$(). This function facilitates the transfer of the previously requested ether amount to the borrower upon successful funding of the loan. Another crucial function that requires implementation is makePayment(uint amount), which borrowers utilize to submit payments towards the loan. The success of this method is contingent on the sender (msg.sender) having adequate funds, and it is also designed to modify the borrower's credit score as part of its execution. Lastly, the withdraw(uint amount) function empowers lenders to withdraw funds from the account, encompassing the invested amount along with accrued interest.

VI. CONCLUSION

This lending system implements the Autonomous Lending Organization on Ethereum with Credit Scoring as outlined by T. Austin et. al in [1]. The purpose of the project in this semester was to both understand the system's intricacies and the challenges of a Decentralized finance (DeFi) product that is built on the Ethereum platform. Each of the four aforementioned deliverables contributed to these goals.

Deliverable 1 laid the foundation for programming smart contracts for the Ethereum blockchain platform using Solidity. In this 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.

Deliverable 2 enabled me to learn how to 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. Deliverable 3, the primary contributor to this project, implementing 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.

Deliverable 4 explores 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.

This project and the project next semester are both aimed at building and testing the Lending system that includes contracts like Credit Bureau, Registration, and Loan. This preliminary report aims to define the components of the system. In the next semester, I will implement the details of

the Loan and Credit Bureau Smart Contract, which matches borrowers and lenders, tracks repayments, and maintains credit scores for registered users.

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] 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.

[4] 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

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