Decentralized Lending in Ethereum Cryptocurrency

SHASHVAT SRIVASTAVA

February 2018
Abstract

The rise of decentralized virtual currencies such as Bitcoin necessitates the development of economic infrastructure. Blindly using existing centralized systems for decentralized currencies is highly undesirable. The goal of this project was to create a peer-to-peer lending protocol to allow borrowers to take out loans and investors to consistently make returns on loans they fund. First, a novel protocol for peer-to-peer lending with smart contracts was created. The protocol focuses on the storage of a borrower’s credit history, and the protocol is flexible enough to support numerous types of loans. Credit history is tied to the identity of the borrower, and verification authorities attest that the borrower exists and has the claimed identity. A trusted “calculator” assesses the credit risk of the borrower; investors use scores computed by calculators to decide which loans to invest in. The security of this protocol was analyzed and tested. Several methods to detect abuse of power by verification authorities were proposed and analyzed. Finally, an example implementation was created in the Solidity smart contract language for the Ethereum cryptocurrency.
4.2 Protocol Overview ........................................ 15
4.3 Protocol Analysis ........................................ 16

5 Results ..................................................... 19
5.1 Conclusions ............................................. 20

References .................................................. 21

6 Appendix .................................................. 22
6.1 Limitations and Assumptions .......................... 22
6.2 Protocol Implementation ............................... 23
6.3 Simulations ............................................. 39
   6.3.1 Simulation Code For Method 1 ................. 39
   6.3.2 Simulation Code For Method 2 ................. 46
1. Literature Review

1.1 Bitcoin: A Peer-to-Peer Electronic Cash System

(Nakamoto, 2008)

In 2008, Satoshi Nakamota introduced the idea of Bitcoin, a decentralized, peer-to-peer electronic cash system. Fundamentally, Bitcoin is ledger of transactions.

**Identification**  Each user is identified by an address; each address is the hash of a public key. Transactions are signed by the private key of the corresponding public key; funds are sent from address to address.

**Transactions**  To send funds, a user must create two proofs. First, a user must show where he acquired the funds he is sending. The only way to receive funds is through a transaction, so the user provides a pointer to the transaction that he received the funds from. Second, a user must prove that he controls the source of funds. Funds are held in addresses that corresponding to public keys. The user must provide a digital signature using the private key to the public key to authorize the transaction. The user signs the entire transaction to prevents its internal contents from being modified.

Upon creation, transactions are broadcasted to the peer-to-peer network, and are added to the ledger.

**Double Spending**  The transaction scheme prevents the theft of coin; however, it does not protect against double-spending. Suppose Alice sends a coin to Bob; later on, she sends
Figure 1.1: When Owner 1 creates a transaction, he provides a reference, or pointer, to the origin of his funds. He also produces a digital signature to prove that he has authorization to spend said funds.

the same coin to Carl as well. Broadcasting both transactions would create confusion as to which transaction is legitimate. Bitcoin uses proof-of-work as a consensus protocol to prevent this type of attack.

Proof-of-work Instead of transactions, the blockchain now communicates in blocks, which are groups of transactions bundled together. To prevent a malicious entity from taking over the blockchain, each block comes with a proof-of-work, performed by a miner. A block is only accepted if the hash of the block satisfies certain restraints. Each block has a nonce (see Figure 1.2) that is modified repeatedly in order to create a block that satisfies the restraints. This process is known as mining. Miners incentivised to mine by money: the creator of a block is allowed to add one additional transaction that mints a fixed amount of funds in their name.

Whenever a miner produces a block, the block is added to the blockchain. Each block references the block before it. At any moment, there may be multiple chains of blocks branching from each other. To create consensus, miners only mine upon the largest chain and discard smaller chains.
Figure 1.2: Each block contains a nonce for the proof-of-work and a pointer to the block before it.

1.2 Ethereum

(Buterin et al., 2013) (Wood, 2014)

Bitcoin provides a simple stack based programming language, Script, that can be used to create features such as multisignatures and escrow transactions. Although Bitcoin’s Script can be considered to be a programming language, it is not Turing complete and its capabilities are limited. The main reason for this is that a Turing complete blockchain language can be easily taken advantage of. A simple infinite loop creates great complications in detecting which computations to perform. As an alternative system to Bitcoin, Ethereum provides Turing complete smart contracts that run on the Ethereum Virtual Machine (EVM).

1.2.1 Ethereum Details

Ethereum can be considered to be a state-transition system. The Ethereum system is composed of objects known as accounts. There are two main types of accounts: “externally owned accounts” and “contract accounts.” Externally owned accounts are controlled by private keys while contract accounts are fully autonomous and governed by their contract code. Both types of accounts have the ability to send messages and create new accounts. A message is essentially a transaction; messages transfer Ether, the currency for Ethereum, from account to account. If a contract account receives a message, its code will trigger in order to determine what actions will happen next.
1.2.2 Ethereum Fees and Gas

Ethereum’s solution to the “infinite loop/excessive computation problem” is to charge a fee for computation. Each message that is sent specifies an amount of Gas for the message. Each computational step costs some Gas. If the Gas is completely spent before the code is finished, the execution completely stops and all changes are reverted. Otherwise, any leftover gas is returned to the message sender. This prevents infinite loops in any smart contracts, as the Gas supply sent in a transaction is finite, forcing all computation to come to an end eventually. This also prevents malicious users from attacking the blockchain via excessive computation since the amount of computation is proportional to the amount paid.

1.3 Other Cryptocurrencies and Protocols

1.3.1 Siacoin

Sia is a decentralized cloud storage platform. Rather than renting cloud storage from a centralized platform, Sia users rent storage from each other.

A storage provider, also known as a host, agrees to store a client’s data. The storage provider periodically provides proof of their continued storage. The storage provider is compensated for each proof they provide, but is penalized if they miss a proof. The proofs are publicly verifiable and are stored on the blockchain. Thus, users do not have to manually verify each proof.

1.3.2 Monero

(Noether, Mackenzie, & Team, 2016) (Sun, Au, Liu, & Yuen, 2017)

Monero is a cryptocurrency based on the CryptoNote protocol (van Saberhagen, 2013). CryptoNote provides users with anonymous payment system using a ring signature technology that allows users to sign messages on behalf of a group. Ring signatures obfuscate the
transactions via *mixins*. Each output describes money that is waiting to be spent by its owner. Each input to a transaction *references* a list of previous outputs (the *mixins*), only one of which is actually being spent. This way the blockchain does not reveal the transaction output that is actually being spent. Additionally, the ring signatures are *linkable* to prevent an output from being spent twice. Monero provides users with persistent cryptographic identities called *addresses*. In order to hide the user identity, Monero uses *Stealth Addresses* to blind the *address* from blockchain observers.

In 2016, Monero deployed ring-confidential transactions (*ringCT*) (Noether et al., 2016), which hide the amount of money involved in a transaction. In 2017, Monero updated the ringCT protocol to decrease the memory size of the protocol (Sun et al., 2017).

### 1.3.3 Tumblebit

Tumblebit is a payment protocol for anonymous payments. The Tumblebit protocol is fully compatible with Bitcoin. The Tumblebit protocol operates through a *Tumbler* that can be used to make unlinkable payments. The importance of Tumblebit lies in that the Tumbler is *untrusted*: the Tumbler can neither steal funds nor deanonymize users.

(Heilman, Alshenibr, Baldimtsi, Scafuro, & Goldberg, 2016)

### 1.4 Lending

With the introduction of decentralized smart contracts, several protocols for decentralized lending have been proposed.

### 1.4.1 Ethlend

Ethlend provides facilities for decentralized lending, namely collateralized lending. The protocol makes use of Ethereum ERC-20 compliant tokens as collateral.

A borrower starts the protocol by creating a loan request that specifies the principle,
interest rate, collateral, and other factors. The borrower then sends the specified coins to the contract. A lender can choose to fund the contract, which initiates the loan.

When the allotted time to repay the loan has passed, the loan contract performs on of two tasks:

- If the borrower has fully repaid their loan, the loan contract returns the collateralized tokens to the borrower.
- Otherwise, the loan contract transfers the tokens to the lender.

The protocol also allows for features such as On-demand learning, which allows lenders to create the loan contract and specify the principle and interest rate. Borrowers then initiate the loan by sending ERC-20 tokens to the loan contract. The Ethlend whitepaper also discusses the possibility of using the Ethereum Name Service as collateral as well.

(Ethlend, 2017)

1.4.2 Dharma Version 1

The Dharma whitepaper makes a first attempt and creating a protocol for decentralized lending. The main difference between Dharma and Ethlend is that the Dharma paper focuses on uncollateralized lending, rather than collateralized lending.

The Dharma protocol consists of three main “layers.”

1. Loan Layer: Each loan consists of a central smart contract that controls the main logic of the loan.

2. Origination Layer: an origination layer contract matches borrowers with lenders.

3. Risk-Assessment Layer: A RAA, or Risk-Assessment Attestor, assesses the authenticity of a borrowers identity and assesses the credit risk.

(Hollander, 2017b)
1.5 Dharma Version 2

(Hollander, 2017a) The Dharma version 2 protocol improves upon the previous Dharma protocol. The protocol consists of four main types of actors: debtors, underwriters, relayers, and creditors. A basic overview:

1. Debtors: Debtors are parties that borrow assets and funds.

2. Creditors: Creditors are parties that lend assets and funds.

3. Underwriters: Underwriters negotiate terms of a loan with a debtor, forward the loan to “relayers”, and commit to the likelihood that the loan will end in default.

4. Relayers: Maintain an “orderbook” of loans that creditors can invest in. Relayers essentially connect the creditors to the loans.
2. Introduction

2.1 Motivation

Decentralized currencies such as Bitcoin are radical new innovations in the world. Despite the restraints created by the nature of decentralized currencies, cryptographic techniques have allowed the creation of surprisingly versatile coins. For example, Monero, based on the CryptoNote protocol, enables users to create transactions that hide the value, sender, and receiver of a transaction.

Suppose, in the future, that decentralized currencies completely replace traditional currencies, but that the current financial system of large central banks issuing loans remains the same. It would be quite redundant if people were to use a decentralized cryptocurrency, but then be forced to rely on centralized banks to store funds so that they could provide liquidity for loans. Therefore, a decentralized peer-to-peer lending protocol in necessary for if cryptocurrencies are to ever reach mass-adoption.

2.2 Project Overview

Engineering Problem: The raison d’être of a cryptocurrency is decentralization. However, most facilities for cryptocurrency lending are highly centralized.

Engineering Goal: The goal of this project was to create a decentralized lending protocol that allows borrowers to take out loans, and investors to consistently make returns.
**Procedure**  First, an initial protocol was drafted and formalized in the form of pseudocode. Next, an implementation of the protocol was created in the Solidity programming language for the Ethereum cryptocurrency. The protocol was updated and improved several times, and the implementation was updated accordingly. The protocol is discussed in chapter 4; the full implementation is in the appendix. Finally, simulations were conducted to demonstrate the validity of several key assumptions.
3. Methodology

The protocol is explained in chapter 5. A reference implementation of the protocol was implemented for the EVM (Ethereum Virtual Machine) in the Solidity programming language. The reference implementation was also informally tested in a private Ethereum test network. Theoretically, the protocol could be implemented in a different decentralized smart contract framework, although the protocol was designed with Ethereum in mind.

As discussed in chapter 4, verifiers must be trusted for the security of the protocol. Therefore, it is critical to be able to detect when a verifier is lying or being dishonest. Two methods of detecting verifiers are proposed.

Method 1 Most verifiers will be under close scrutiny of the public, as their job is to do a task under trust. Inevitably, some dishonest verifiers will make mistakes and get caught. Specifically, consider the case when an account, for whatever reason, is discovered to be a fraudulent account. Then, every verifier that verified that account must be dishonest. Furthermore, suppose that over half of an account’s (not necessarily fraudulent) verifiers are dishonest. Then, that account is most likely a fake account. Using these two observations, a series of “leaked” fake accounts can result in a chain reaction of dishonest verifiers being detected.

Method 2 Method 2 uses Method 1, but is significantly different. Method 1 requires fake accounts to be leaked in some manner, and implicitly assumes that this information is correct. On the other hand, this heuristic uses on blockchain verification data to determine the fake
accounts. The chain reaction method is then applied along with this method to detect the dishonest verifiers.

Specifically, it is assumed to be unusual for any two verifiers to have verified any account multiple times. On the other hand, dishonest verifiers must work together to successfully create a fraudulent account. Thus, the rate of sharing verifications will be much higher for dishonest verifiers than for honest verifiers.

Let the verifiers be denoted as \{1, 2, ..., n\}. Let \(\alpha_{i,j}\) denote the number of borrowers verified by both verifier \(i\) and verifier \(j\). Then, define the score \(s_i\) of borrower \(i\) as

\[
\frac{\sum_{j=1}^{n} \alpha_{i,j}}{M}
\]

, where

\[
M = \sum_{j=1}^{n} \alpha_{i,j}
\]

Finally, let the score \(d\) of a borrower be

\[
d_k = \sum_{i \in V_k} s_i
\]

, where \(V_k\) is the set of verifiers for borrower \(k\). Then, the outliers, or the borrowers with the highest scores, will most likely be fake. One can then run the chain-reaction detection method, using the calculated fake borrowers rather than leaked borrowers.

Monte Carlo simulations were conducted to test both methods. A fake “verification-graph” was first generated according to Assumption 5. The detection methods were then run, and the results are shown in chapter 5. The code for these two methods can be found in the appendix.
4. Lending Protocol

4.1 Problems with Current Protocols

In order to properly service borrowers, it is necessary to provide different types of loans. Although Ethlend (Ethlend, 2017) provides a lending service, its only provides a platform for collateralized loans. However, many loans are also uncollateralized. Dharma (Hollander, 2017b, 2017a) provides a platform for both collateralized and uncollateralized loans. In the second iteration of their protocol, “underwriters” assess credit risk, negotiate the loan terms, and generally “service” the loan. However, the Dharma protocol never mentions how the underwriter can assess the credit risk. The concern is that underwriters are forced to get data from centralized services in order to assess credit risk, much like credit bureaus today. Therefore, it a protocol that focuses on the storage of a borrower’s credit history was created.

4.2 Protocol Overview

The problem with decentralized credit history storage is that a borrower can simply create multiple accounts. This is prevented by use of verification authorities (or simply “verifiers”). From Assumption 1, each person can be uniquely classified by parameters $X_1, X_2, ..., X_k$. From Assumption 2, an honest verifier will be correctly able to deduce the parameters $X_1, X_2, ..., X_k$. When a borrower makes a borrowing account, called a proxy borrower account, he or she specifies their unique identifier, $I = H(X_1||X_2||...||X_k)$, where $H$ is a hash function. A verifier, or most likely multiple verifiers, agree to “verify” the
borrower in exchange for a monetary fee. A full analysis of this situation is conducted in section 4.3. The borrower uses the proxy borrower account to interact with “calculators”. Calculators assess credit risk and negotiate loan terms with the borrower. In exchange, calculators receive a portion of the interest paid by the borrower. The calculator is able to do this, as the entire credit history of the borrower is stored on the blockchain. The calculator interacts with borrowers and loans via a proxy calculator account, which allows loan investors to additionally judge the accuracy of a calculator’s predictions. Finally, verifiers also hold a proxy verification account to make it easy to see which borrowers the verifier verified.

The implementation of the protocol is in the appendix.

4.3 Protocol Analysis

The goal of this project was to both allow investors to consistently make returns on their investments, and to allow borrowers to freely take out loans of different types. Most importantly, it is important to allow loans to be uncollateralized, since some loans in the current financial world are uncollateralized. The main risk with uncollateralized loans is fraud: an attacker can simply request a loan, and never pay it back. Systems based purely on trust will not fail to protect against these attacks unless the cost of gaining trust is higher than the payoff of the stolen loan. However, if the cost of gaining trust is comparable to the to the amount that might be stolen, it is the borrowers that are hurt, in this case by large fees. Therefore, it is necessary to provide a system that does not work purely on trust.

The solution is to tie the identity of a borrower with that of a person. If implemented correctly, each person will only be able to create one borrowing account in their lifetime. This borrowing account will hold all of their loan data, so if they commit fraud, if will be recorded “for eternity”. They will most likely never be able to (successfully) request a loan again. Of course, there will still be those steal, but they will be a minority of the population.
This technique also prevents entities such as corporations from holding borrowing accounts in the same way; it is “human-centric”.

We now analyze the safety of this method.

Suppose that a borrower has created a proxy borrower account that has been sufficiently verified by trusted verifiers. Suppose that he or she attempts to make a second account. We break this down into the following two cases.

- **Case 1:** The borrower creates the account under an identifier that does not belong to them. Then, the borrower will not be able to convince any verifier to verify them, as per Assumption 2. With no verifications, the borrower will not be able to negotiate with any calculator, and will not get any funding by any investors. The borrower will simply have created a lump of data that will be ignored.

- **Case 2:** The borrower creates the account under their own identifier. Although they might be able to convince the verifier that the identifier belongs to them, the verifier will notice that the identifier already belongs to another verified account. Since Assumption 1 holds, the verifier will recognize that the borrower already has an account, and will refuse to verify the second account.

- **Case 3:** For sake of argument, suppose that a borrower manages to create two accounts with a sufficient number of verifications. First, all verifiers involved will lose trust, as this scenario should never occur in the first place. Furthermore, all calculators and investors will refuse to transact with *either* of the proxy borrower accounts, as they will recognize the fraud.

Note that all of this occurs with a fourth, hidden assumption: Assumption 4.

Therefore, it is impossible, or at least highly unlikely, for a borrower to multiple accounts that receive sufficient verifications to operate. Thus, by Assumption 5, most borrowers will not commit direct fraud, in which they take out a loan with the intention of never paying it back.
The other possibilities of fraud might come from either the investors, the verifiers, or the calculators.

The only participation from investors is to invest in loans; there is no chance of fraud from an investor. A verifier may be able to commit fraud by verifying accounts that they know to be fake. Even though the verifiers are assumed to be honest, detection of this type of fraud is discussed in chapter 3.

A calculator assesses credit risk. Normally, a calculator’s predictive abilities can be judged their previous predictions. A calculator will not be able to artificially inflate its track record by use of fake accounts, as they will not be verified. However, if the calculator is able to gain control, or at least partial control, over multiple verified accounts, the calculator will be able to inflate its predictive history. This type of attack/fraud is left for future work.
5. Results

Figure 5.1 shows the results of the Monte Carlo simulation for the “chain reaction” defense. The number of dishonest verifiers marked as dishonest grows linearly against the number of leaked fake accounts until the end. In most cases, the method was extremely effective. The false-positive was zero for all cases.

Figure 5.2 shows the results of the Monte Carlo simulation for the pairing based analysis (method 2). The method was close to perfect for the test cases with only 100 dishonest
verifiers. The method was ineffective for most of the test-cases with 1000 dishonest verifiers. This is to be expected, as the idea behind method 2 becomes less effective as the number of dishonest verifiers to choose from increases.

![Results of Pairing Based Heuristic](image)

Figure 5.2: DV: Dishonest verifier; HV: Honest verifier. There were 10000 verifiers in total in all cases. There were also 100000, 1000000, and 5000000 borrowers in each “set” (from left to right). The numbers below the bars indicate the portion of borrowers that were fake. Both the number of dishonest verifiers detected, as well number of honest verifiers marked as dishonest, are shown.

### 5.1 Conclusions

A lending protocol was created. The protocol is a clear improvement over existing protocols. The security of the protocol was analyzed and tested. The assumptions behind the protocol were questioned, and it was determined that they would most likely hold true. Finally, an implementation of the protocol was created.

Future work could include improvements to the verifier detection algorithm, instantiations of the parameters assumed in Assumption 3, and detection/prevention of fraud from “calculators”.

20
References


6. Appendix

6.1 Limitations and Assumptions

Assumption 1 Each human can be uniquely identified by a set of parameters $X_1, X_2, ..., X_k$ that do not change. These parameters can range from simple items, such as name (at date of birth), birthday, and gender, to more complex parameters based on biometrics.

Assumption 2 Either all verification authorities are honest, or dishonest verification authorities are known and can be ignored.

Assumption 3 A honest borrower will be able to convince an honest verifier that parameters $X_1, X_2, ..., X_k$ belong to him. A dishonest borrower will be unable to convince an honest verifier that parameters $X_1, X_2, ..., X_k$ belong to the them, unless parameters $X_1, X_2, ..., X_k$ actually to belong to the borrower.

Assumption 4 Out of the set of all honest versifiers, all honest entities will trust and know of, the majority of the verifiers.

Assumption 5 The verifiers for an honest borrower can be treated as if they were chosen randomly from the set of all verifiers. The verifiers for a fake account can be treated as if they were chosen randomly out of the set of all dishonest verifiers.
**Assumption 6** All borrowers need exactly $k$ verifications to be trusted, where $k$ is a positive integer.

### 6.2 Protocol Implementation

```solidity
pragma solidity ^0.4.10;

contract ProxyAccount {
    address public owner;
    ProtocolManager internal pm;
    uint256 internal fakeBalance;

    function () payable public {
        fakeBalance += msg.value;
    }

    function ProxyAccount(address _owner) public {
        owner = _owner;
        pm = ProtocolManager(msg.sender);
    }

    function sendTo(address dest, uint256 value) onlyOwner() public {
        require(value <= fakeBalance);
        fakeBalance -= value;
        dest.transfer(value);
    }
}
```
modifier onlyOwner() {
    require(msg.sender == owner);
-;
}
modifier onlyProxyAccount(uint8 bvc, address account) {
    require(pm.isProxyAccount(bvc, account));
-;
}

contract ProxyBorrower is ProxyAccount {
    bytes32 public hashID;
    address[] public verifiers;
    OneTimeAccount[] public loans;

    function ProxyBorrower(address _owner, bytes32 h) ProxyAccount (_owner) public {
        hashID = h;
    }

    function requestVerification(ProxyVerifier verifier, uint256 _new) onlyOwner() onlyProxyAccount(pm.VERIFIER(), verifier) public {
        require(_new != 0);
        uint256 old = verifier.requests(this);
require(fakeBalance + old >= _new);
fakeBalance = fakeBalance + old - _new;
ProxyVerifier( verifier ).makeRequest(_new);
}

function addVerification() onlyProxyAccount(pm.VERIFIER(), msg.sender) public {
    verifiers.push(msg.sender);
    // strange error here
    uint256 val = ProxyVerifier(msg.sender).requests(this);
    msg.sender.transfer(val);
}

function createOneTimeAcc(address payer, address calculator)
    public onlyProxyAccount(pm.CALCULATOR(), calculator)
    onlyOwner() returns (OneTimeAccount) {
    OneTimeAccount acc = pm.createOneTimeAcc(payer, calculator);
    loans.push(acc);
    return acc;
}

contract ProxyVerifier is ProxyAccount {

function ProxyVerifier(address _owner) ProxyAccount(_owner) 
    public { }

event Info(string str);
function addInfo(string str) public onlyOwner() {
    Info(str);
}

mapping(address => uint256) public requests;
mapping(address => bool) public finished_requests;
function makeRequest(uint256 val) onlyProxyAccount(pm.BORROWER(), msg.sender) public {
    requests[msg.sender] = val;
}

function addVerification(address borrower) onlyOwner() 
    onlyProxyAccount(pm.BORROWER(), borrower) public {
    require(requests[borrower] != 0 && !finished_requests[borrower]);
    ProxyBorrower(borrower).addVerification();
    requests[borrower] = 0;
    finished_requests[borrower] = true;
}

contract ProxyCalculator is ProxyAccount {

function ProxyCalculator(address _owner) ProxyAccount(_owner)
    public { }

address[] loans;
mapping(address => bool) tf;

function changeScore(OneTimeAccount acc, uint256 score) public
    onlyOwner() {
        require(pm.isProxyAccount(pm.ONETIME(), acc));
        if (!tf[acc]) {
            tf[acc] = true;
            loans.push(acc);
        }
        acc.changeScore(score);
    }

function changeShares(OneTimeAccount acc, uint256 val) public
    onlyOwner() {
        require(pm.isProxyAccount(pm.ONETIME(), acc));
        if (!tf[acc]) {
            tf[acc] = true;
            loans.push(acc);
        }
        acc.changeShares(address(this), val);
    }
function changeEthPerShare(OneTimeAccount acc, uint256 _ethPerShare) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    if (!tf[acc]) {
        tf[acc] = true;
        loans.push(acc);
    }
    acc.changeEthPerShare(_ethPerShare);
}

function changeTerms(OneTimeAccount acc, LoanTerms newterms) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    if (!tf[acc]) {
        tf[acc] = true;
        loans.push(acc);
    }
    acc.changeTerms(newterms);
}

function changeFundingPeriod(OneTimeAccount acc, uint256 _fundingperiod) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    if (!tf[acc]) {
        tf[acc] = true;
        loans.push(acc);
    }
}
acc.changeFundingPeriod(_fundingperiod);

function acceptTerms(OneTimeAccount acc) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    if (!tf[acc]) {
        tf[acc] = true;
        loans.push(acc);
    }
    acc.acceptTerms();
}

define mapping(address => uint256) predictionbuildingscores;
uint256 constant public OUTOF =
    1000000000000000000000000000000000000000000000000000000000;

function unblindPredictionBuilding(OneTimeAccount acc, uint256 score) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    acc.makeunblindPrediction(score);
    predictionbuildingscores[acc] = score;
}

function blindPredictionBuilding(OneTimeAccount acc, bytes32 score) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    acc.makeblindPrediction(score);
function unblindPrediction(OneTimeAccount acc, uint256 score, uint256 mask) public onlyOwner() {
    require(pm.isProxyAccount(pm.ONETIME(), acc));
    require(acc.blindPredictions(this) == keccak256(OUTOF * mask + score));
    require(acc.terms().isDone() || acc.terms().inDefault());
    predictionbuildingscores[acc] = score;
}

contract ProtocolManager {
    uint8 public constant BORROWER = 0;
    uint8 public constant VERIFIER = 1;
    uint8 public constant CALCULATOR = 2;
    uint8 public constant ONETIME = 3;

    mapping(address => bool[4]) public accounts;
    mapping(bytes32 => address[]) public IDCollisions;

    function isProxyAccount(uint8 bvc, address addr) view public returns (bool) {
        return accounts[addr][bvc];
    }
}
function createBorrower(bytes32 hashID) public returns (ProxyBorrower) {
    ProxyBorrower newaccount = new ProxyBorrower(msg.sender, hashID);
    accounts[newaccount][BORROWER] = true;
    Creation(newaccount, BORROWER, now);
    IDCollisions[hashID].push(msg.sender);
    return newaccount;
}

function createVerifier() public returns (ProxyVerifier) {
    ProxyVerifier newaccount = new ProxyVerifier(msg.sender);
    accounts[newaccount][VERIFIER] = true;
    Creation(newaccount, VERIFIER, now);
    return newaccount;
}

function createCalculator() public returns (ProxyCalculator) {
    ProxyCalculator newaccount = new ProxyCalculator(msg.sender);
    accounts[newaccount][CALCULATOR] = true;
    Creation(newaccount, CALCULATOR, now);
    return newaccount;
}

function createOneTimeAcc(address payer, address calculator) public returns (OneTimeAccount) {
    require(isProxyAccount(BORROWER, msg.sender) &&
        isProxyAccount(CALCULATOR, calculator));
OneTimeAccount acc = new OneTimeAccount(msg.sender, payer, 
calculator);
accounts[acc][ONETIME] = true;
Creation(acc, ONETIME, now);
return acc;
}

functiongetIDCollisionsLength(bytes32 id) public view returns 
(uint256) {
    return IDCollisions[id].length;
}

event Creation(
    address indexed addr,
    uint8 indexed bvc,
    uint256 indexed time
);

}
function triggerDefund() public;
}

contract OneTimeAccount {
    ProtocolManager public pm;

    address public borrower;
    address public payer;
    address public calculator;
    bool public baccept = false;
    bool public caccept = false;

    uint8 public constant NEGOTIATION = 0;
    uint8 public constant ACCEPT = 1;
    uint8 public constant FUNDING = 2;
    uint8 public constant PAYMENT = 3;
    uint8 public constant DEFUND = 4;

    uint8 public stage = 0;

    function OneTimeAccount(address _borrower, address _payer,
        address _calculator) public {
        pm = ProtocolManager(msg.sender);
        borrower = _borrower;
        payer = _payer;
        calculator = _calculator;
    }
uint256 public score = 0;
//10^36, or 10^18 ether, good enough precision?
uint256 constant public OUTOF =
    10000000000000000000000000000000000000000000;

function changeScore(uint256 _score) public {
    require(msg.sender == calculator && stage == NEGOTIATION);
    score = _score;
}

bool public allowPredictionBuilding = true;

function changePredictionBuilding(bool tf) public negotiation
    () {
    allowPredictionBuilding = tf;
}

mapping(address => uint256) public unblindPredictions;
function makeunblindPrediction(uint256 _score) public {
    require(pm.isProxyAccount(pm.CALCULATOR(), msg.sender));
    require((stage == NEGOTIATION || stage == ACCEPT || stage
    == FUNDING));
    unblindPredictions[msg.sender] = _score;
}
mapping(address => bytes32) public blindPredictions;

function makeblindPrediction(bytes32 _score) public {
    require(pm.isProxyAccount(pm.CALCULATOR(), msg.sender));
    require((stage == NEGOTIATION || stage == ACCEPT || stage == FUNDING));
    blindPredictions[msg.sender] = _score;
}

mapping(address => uint256) public shares;

uint256 totalShares = 0;

uint256 public ethPerShare = 1;

function changeShares(address addr, uint256 val) public negotiation() {
    require(addr == calculator || addr == address(this));
    shares[addr] = val;
}

function changeEthPerShare(uint256 _ethPerShare) public negotiation() {
    ethPerShare = _ethPerShare;
}

LoanTerms public terms;

function changeTerms(LoanTerms newterms) public negotiation() {
    terms = newterms;
}
uint256 public fundingPeriod;

function changeFundingPeriod(uint256 _fundingperiod) public
    negotiation()
    {
        fundingPeriod = _fundingperiod;
    }

function acceptTerms() public {
    require((stage == NEGOTIATION || stage == ACCEPT);
    if (msg.sender == calculator)
        caccept = true;
    if (msg.sender == payer)
        baccept = true;
    if (baccept || caccept) {
        stage = ACCEPT;
    }
    if (baccept && caccept && terms.preCondition()) {
        stage = FUNDING;
    }
}

mapping(address => uint256) public balances;

address[] public payers;

function fundLoan() public payable {
    require((stage == FUNDING && msg.value % ethPerShare == 0);
require(now <= fundingPeriod);

uint256 numShares = msg.value / ethPerShare;

if (shares[msg.sender] == 0)
    payers.push(msg.sender);

if (numShares > shares[this]) {
    balances[msg.sender] += ethPerShare * (numShares - shares[this]);
    numShares = shares[this];
}

shares[this] -= numShares;
shares[msg.sender] += numShares;

if (shares[this] == 0) {
    balances[this] = 0;
    totalShares = 0;
    for (uint256 i = 0; i < payers.length; ++i) {
        balances[this] += shares[payers[i]] * ethPerShare;
        totalShares += shares[payers[i]];
    }
    stage = PAYMENT;
    totalShares += shares[calculator];
}

function triggerDefund() public payable {
    require((stage == FUNDING && fundingPeriod < now));
    stage = DEFUND;
    for (uint256 i = 0; i < payers.length; ++i) {
        balances[payers[i]] += ethPerShare * shares[payers[i]];
    }
    terms.triggerDefund();
}

function makePayment() public payable {
    require((stage == PAYMENT && msg.sender == payer && msg.value % totalShares == 0));
    for (uint256 i = 0; i < payers.length; ++i) {
        balances[payers[i]] += (msg.value / totalShares) * ethPerShare * shares[payers[i]];
    }
    balances[calculator] += (msg.value / totalShares) * ethPerShare * shares[calculator];
    terms.makePayment(msg.value);
}

function withdraw(uint256 amount) public {
    require(balances[msg.sender] > amount);
    balances[msg.sender] -= amount;
    msg.sender.transfer(amount);
}
function transferShare(uint256 amount, address addr) public {
    require(amount <= shares[msg.sender]);
    if (shares[addr] == 0)
        payers.push(addr);
    shares[addr] += amount;
    shares[msg.sender] -= amount;
}

modifier negotiation() {
    require(stage == NEGOTIATION && (msg.sender == payer ||
        msg.sender == calculator));
}

6.3 Simulations

6.3.1 Simulation Code For Method 1

#include <iostream>
#include <fstream>
#include <vector>
#include <tuple>
#include <stack>
```cpp
#include <time.h>
#include <random>
#include <iterator>
#include <algorithm>
#include <functional>
#include <sstream>

int seed = 0;

#define _CRT_SECURE_NO_DEPRECATE
using namespace std;
#define MAXVERIFIERS 10000
#define MAXACCOUNTS 5000000

vector<int> bad = {};
vector<int> goodtrack = {};
vector<int> badtrack = {};
int countgood = 0;
int countbad = 0;
vector<int> graph[MAXVERIFIERS];
pair<bool, vector<int>> revgraph[MAXACCOUNTS];

void generate_graph(const int nborrowers, const int ngood,
                     const int nbad, const double lieratio, const int verifneeded) {
    const int nverif = ngood + nbad;
```
mt19937 rng(seed);

for (int i = 0; i < nverif; ++i) graph[i] = {};
for (int i = 0; i < nborrowers; ++i) revgraph[i] = {};
bad = {};

vector<int> nots = {};
for (int i = 0; i < nborrowers; ++i) {
    nots.push_back(i);
}

for (int i = 0; i < nborrowers; ++i) {
    static mt19937 rnd(seed);
    static uniform_real_distribution<double> d(0, 1);
    double rnum = d(rnd);
    bool tf = false;
    int end = nverif;

    if (rnum < lieratio) {
        tf = true, end = nbad;
        bad.push_back(i);
    }
}

vector<int> s(verifneeded);
vector<int> fix(verifneeded * 2);
for (int i = 0; i < verifneeded; ++i) {
int t = uniform_int_distribution<int>(i, end - 1)(rng);
swap(nots[i], nots[t]);
s[i] = nots[i];
fix[2 * i] = i;
fix[2 * i + 1] = t;
}
for (int a : fix) {
    nots[a] = a;
}

revgraph[i] = { tf, s };
for (int a : s)
    graph[a].push_back(i);
}
return;
}

void chain_reaction_calc(const int nborrowers, const int ngood,
const int nbad,
const int verifneeded, const double chainreaccoef) {
vector<int> lst = bad;
random_shuffle(lst.begin(), lst.end());

vector<bool> visited(nborrowers, false);
vector<int> count(nborrowers, false);
vector<bool> dishonest(ngood + nbad, false);

countgood = 0;
countbad = 0;

for (int t : lst) {
    if (visited[t]) continue;
    stack<int> s;
    s.push(t);

    while (!s.empty()) {
        int cur = s.top();
        s.pop();

        if (visited[cur]) continue;
        visited[cur] = true;

        for (int verifier : revgraph[cur].second) {
            if (dishonest[verifier])
                continue;
            dishonest[verifier] = true;

            if (verifier < nbad) ++countbad;
            else ++countgood;
        }
    }
}
for (int n : graph[verifier]) {
    count[n] += 1;
    if (count[n] >=
        verifneeded *
        chainreaccoef && !
        visited[n])
        s.push(n);
}

}

}

}

goodtrack.push_back(countgood);
badtrack.push_back(countbad);
}

}

void solve(tuple<int, int, double, int, double> params, const
char* file) {
    int ngood, nbad, verifneeded;
    double lieratio, chainreaccoef;
    tie(ngood, nbad, lieratio, verifneeded, chainreaccoef)
    = params;

    freopen(file, "a", stdout);
    int arr[] = { 100000, 1000000, 2000000, 3000000,
for (int nborrowers : arr) {
    nborrowers = arr[1];
    countgood = 0;
    countbad = 0;
    goodtrack = {};
    badtrack = {};

    generate_graph(nborrowers, ngood, nbad,
                   lieratio, verifneeded);
    chain_reaction_calc(nborrowers, ngood, nbad,
                        verifneeded, chainreaccoef);

    printf("%d %d %d %d %f %f\n", nborrowers, ngood
              , nbad, verifneeded, lieratio, chainreaccoef
        );
    for (int a : badtrack) cout << a << ' ';
    cout << ' ';
    for (int a : goodtrack) cout << a << ' ';
    cout << "\n";
    break;
}

int main() {


```cpp
seed = time(0);
srand(seed);

auto file = "data/method1.txt";
solve({ 9000, 1000, 0.01, 10, 0.7 }, file);
solve({ 9000, 1000, 0.01, 10, 0.8 }, file);
solve({ 9000, 1000, 0.001, 10, 0.7 }, file);
solve({ 9000, 1000, 0.0001, 10, 0.7 }, file);
solve({ 9900, 100, 0.01, 10, 0.7 }, file);
solve({ 9900, 100, 0.01, 10, 0.8 }, file);
solve({ 9900, 100, 0.001, 10, 0.7 }, file);
solve({ 9900, 100, 0.0001, 10, 0.7 }, file);

return 0;
}

6.3.2 Simulation Code For Method 2

#include <iostream>
#include <fstream>
#include <vector>
#include <tuple>
#include <stack>
#include <time.h>
#include <random>
#include <iterator>
#include <algorithm>
#include <functional>
```

46
```cpp
#include <sstream>

int seed = 0;

#define _CRT_SECURE_NO_DEPRECATE
using namespace std;
#define MAXVERIFIERS 10000
#define MAXACCOUNTS 5000000

int countgood = 0;
int countbad = 0;
vector<int> graph[MAXVERIFIERS];
pair<bool, vector<int>> revgraph[MAXACCOUNTS];

vector<vector<int>> gdata;
vector<pair<int, double>> order;

void generate_graph(const int nborrowers, const int ngood,
                     const int nbad, const double lieratio, const int verifneeded) {
  const int nverif = ngood + nbad;

  mt19937 rng(seed);

  for (int i = 0; i < nverif; ++i) graph[i] = {};
  for (int i = 0; i < nborrowers; ++i) revgraph[i] = {};
```

47
vector<int> nots = {};
for (int i = 0; i < nborrowers; ++i) {
    nots.push_back(i);
}

for (int i = 0; i < nborrowers; ++i) {
    static mt19937 rnd(seed);
    static uniform_real_distribution<double> d(0, 1);
    double rnum = d(rnd);
    bool tf = false;
    int end = nverif;

    if (rnum < lieratio)
        tf = true, end = nbad;

    vector<int> s(verifneeded);
    vector<int> fix(verifneeded * 2);
    for (int i = 0; i < verifneeded; ++i) {
        int t = uniform_int_distribution<int>(i, end - 1)(rng);
        swap(nots[i], nots[t]);
        s[i] = nots[i];
        fix[2 * i] = i;
        fix[2 * i + 1] = t;
    }

    for (int a : fix) {
        // Code here
    }
}
nots[a] = a;

revgraph[i] = { tf, s };  
for (int a : s) 
    graph[a].push_back(i);

return;

void graph_calc(const int nborrowers, const int nverif) {
    vector<double> score = {};
    gdata.resize(nverif, vector<int>(nverif, 0));

    for (int i = 0; i < nborrowers; ++i) {
        for (int a : revgraph[i].second)
            for (int b : revgraph[i].second)
                if (a != b)
                    ++gdata[a][b];
    }

    for (int i = 0; i < nverif; ++i) {
        int sum = 0;
        for (int j = 0; j < nverif; ++j)
            sum += gdata[i][j];
        int sum2 = 0;
        
    }
```cpp
for (int j = 0; j < nverif; ++j)
    sum2 += (gdata[i][j]) * (gdata[i][j]);
score.push_back(((double)sum2) / sum);
}

for (int i = 0; i < nborrowers; ++i) {
    double a = 0;
    for (int c : revgraph[i].second) {
        a += score[c];
    }

    order.push_back({ i, a });
}

sort(order.begin(), order.end(), [](pair<int, double> a , pair<int, double> b) {
    return a.second > b.second;
});
return;

void chain_reaction_calc(const int nborrowers, const int ngood,
const int nbad,
const int verifneeded, const double chainreaccoef) {
    vector<int> lst;
    for (auto p : order) lst.push_back(p.first);
```
vector<bool> visited(nborrowers, false);
vector<int> count(nborrowers, false);
vector<bool> dishonest(ngood + nbad, false);

countgood = 0;
countbad = 0;

int counter = 0;
for (int t : lst) {
    //if (counter >= 0.01 * nborrowers)
    //    break;
    if (counter >= 100) break;
    ++counter;
    if (visited[t]) continue;
    stack<int> s;
    s.push(t);

    while (!s.empty()) {
        int cur = s.top();
        s.pop();

        if (visited[cur]) continue;
        visited[cur] = true;

        for (int verifier : revgraph[cur].second) {
            if (dishonest[verifier])
        }
continue;
dishonest[verifier] = true;

if (verifier < nbad) ++countbad;
else ++countgood;

for (int n : graph[verifier]) {
    count[n] += 1;
    if (count[n] >=
        verifneeded *
        chainreaccoef && !
        visited[n])
        s.push(n);
}

void solve(tuple<int, int, double, int, double> params, const char* file) {
    int ngood, nbad, verifneeded;
    double lieratio, chainreaccoef;
}
```c
tie(ngood, nbad, lieratio, verifneeded, chainreaccoef) = params;

freopen(file, "a", stdout);

int arr[] = { 100000, 1000000, 2000000, 3000000, 4000000, 5000000 };
for (int nborrowers : arr) {
    countgood = 0;
    countbad = 0;
    order = {};
    gdata = {};

    generate_graph(nborrowers, ngood, nbad, lieratio, verifneeded);
    graph_calc(nborrowers, ngood + nbad);
    chain_reaction_calc(nborrowers, ngood, nbad, verifneeded, chainreaccoef);

    printf("%d %d %d %d %f %f\n", nborrowers, ngood, nbad, verifneeded, lieratio, chainreaccoef);
    printf("%d %d\n\n", countbad, countgood);
}
```
int main() {
    seed = time(0);
    srand(seed);

    auto file = "data/method22.txt";
    solve({ 9000, 1000, 0.01, 10, 0.7 }, file);
    solve({ 9000, 1000, 0.001, 10, 0.7 }, file);
    solve({ 9000, 1000, 0.0001, 10, 0.7 }, file);
    solve({ 9900, 100, 0.01, 10, 0.7 }, file);
    solve({ 9900, 100, 0.001, 10, 0.7 }, file);
    solve({ 9900, 100, 0.001, 10, 0.7 }, file);

    return 0;
}