Autonomous Lending Organization on Ethereum with Credit Scoring

1st Thomas H. Austin*, 2nd Katerina Potika*, and 3rd Chris Pollett*
*Department of Computer Science
San José State University, San Jose, CA, United States
Email: thomas.austin@sjsu.edu, katerina.potika@sjsu.edu, chris@pollett.org

***Abstract*—We propose the Autonomous Lending Organization on Ethereum (ALOE) system, which enables unsecured borrowing of funds on Ethereum. We incorporate a credit scoring approach that extends in the DeFi world the one that is used by traditional banks in order to quantify the risk of a borrower defaulting on a loan. As part of the loan process, first, we have a registration phase, where a notary verifies the real identity of a borrower and delegates to a set of auditors the task of storing a share of the real identity of a borrower to an Ethereum account while preserving anonymity. In the next phase, the Credit Bureau Smart Contract connects lenders to borrowers and updates credit scores. We automatically compute and update credit scores on-chain using the k-nearest neighbors algorithm.**

***Index Terms*—Credit scores, Decentralized Finance, Decentralized Autonomous Organization, Loan, k-nearest neighbors algorithm, cold start problem, smart contracts**

I. Introduction

Cryptocurrencies are an important, relatively new medium of exchange. These currencies offer some advantages over traditional currencies. They usually operate without relying on a central authority such as a government. They often have built-in to their infrastructure the ability to create smart contracts that can be algorithmically carried out and verified.

One important class of real-world contracts is the loaning of money from one individual or group of individuals to another. In traditional systems, a bank serves as a middle man. It decides the risks it will take when it loans money, and it profits on the difference between the interest it pays its depositors versus the interest it charges to its borrowers. Typically, the bank uses a credit score provided by a trusted credit bureau to measure the credit worthiness of borrowers.

While loan systems also currently exist in the cryptocurrency space, the anonymous nature of most blockchain-based cryptocurrencies limits the types of loans to either *micro-loans*, where money is borrowed and returned in the same block, or to collateralized loans where a user secures their loan by staking another type of cryptocurrency asset. Since tracking a user's identity is difficult, lenders do not take into consideration the credit history of that user.

In this paper, we seek to track a user's credit score and to tie their cryptocurrency account to their real-world identity, but without sacrificing the anonymity of well-behaved borrowers. With this design, lenders can offer more generous terms to borrowers, knowing that they can legally pursue the real-world borrower if they default on their loan.

We design our system to work on the Ethereum blockchain [1]. We include an off-chain notary that ties social-security numbers to Ethereum addresses, and which assigns an initial credit score derived from the user's real-world FICO score (discussed in Section II-A). The real-world identity is divided into secret shares and stored with third-party, off-chain *auditors*, and the notary then forgets the association between the user's Ethereum address and their identity. Thus, as long as the user does not violate the terms of any loans they receive, their anonymity is preserved.

Once the identity is established, a CreditBureau smart contract matches borrowers and lenders, tracks the repayments, and maintains credit scores for registered users.

In particular, we propose a system called the Autonomous Lending Organization in Ethereum (ALOE)[1], that allows a borrower to create a loan application on the blockchain to which investors can contribute capital until the loan funds, obviating the need of a bank. If the loan funding quantity is not reached within a window of time, the funds are returned to the investors. Otherwise, if the loan is funded, the borrower receives the funds and is expected to make regular payments with interest.

To make our cryptocurrency based loan process successful, we need to propose mechanisms that solve many of the challenges that a traditional bank faces in deciding whether to loan to a borrower and with what risk. One of these is that the lenders need to know the borrowers likelihood of repaying their debt. This likelihood is influenced by the borrowers' total outstanding debts, and to know this, the lender needs to establish some facts about the identity of the borrower in such a way that makes it hard for the borrower to fool the system by creating false identities. In the cryptocurrency setting these are ideally established in an anonymizing way, such as via a zero knowledge proof and homomorphic encryption so as to maintain privacy. As the digital and real-world and their finances are inextricably intertwined, any mechanism for estimating the likelihood of repaying debts must allow for a way to incorporate measurements of real world debt. From the perspective of investors the process of investing should be as easy as possible. In the case of a traditional bank, one merely deposits money in the bank; a depositor has some choice in the kind of bank account and the interest rate that they receive.

[1] https://github.com/taustin/cryptoCreditBureau/

We propose two main mechanisms to solve these problems for cryptocurrencies. The first is an identity and credit bureau system for the blockchain that solves the problem of measuring the likelihood of loans to be repaid. The second is a mechanism for categorizing loan smart contracts with similar properties and aggregating these into a larger contract that an investor can choose to invest in and receive payments from.

We now discuss the organization of the rest of the paper. In Section II, we provide some background on a traditional way to obtain credit scores and an overview of the decentralized finance (DeFi) market. In Section III we present our mechanism by establishing a pseudo-anonymous identity with a crypto credit bureau. In Section IV we describe the credit score computations. Finally, we conclude in Section V.

II. Preliminaries and Related Work

Bitcoin [2] first introduced the world to the power of the blockchain for maintaining a ledger without a central authority. It included the Bitcoin Script language for its transactions, giving it a limited degree of flexibility in the type of transactions it could conduct. Ethereum [1] builds on Bitcoin's ideas to include a quasi-Turing-complete virtual machine to the blockchain. Solidity [3] is currently the most popular language for writing Ethereum smart contracts [4].

A few features of both Ethereum and Solidity need to be emphasized in order to better understand our design. In Ethereum, there are two types of accounts: an *externally owned account* (EOA) is associated with a public key, and hence owned by an external user, while a *contract account* is associated with some code. Critically, all transactions must be initiated by an EOA. Within Solidity, the EOA that initiated a transaction is identified by the variable `tx.origin`. The caller of a smart contract (which may be either an EOA or a contract account) is identified by `msg.sender`. Transactions may send data and/or ether; if ether is sent, the amount of ether transferred is specified by `msg.value`.

A. Traditional Credit Scores

We describe a common way of calculating a credit score for a traditional loan. After our description, we briefly point out the arithmetical operations that it contains and how they are amenable to homomorphic encryption.

The Fair Isaac corporation (now called FICO) is a data analytics firm responsible for some of the more popular credit scores. The FICO Score 8, often used for home loans, consists of the following five components (for now we ignore how these are calculated): Payment History (PH), Debt Burden (DB), Length of Credit History (LoCH), Types of Credit used (ToC), and Recent Credit Searches (RCS). Using these components the following weighted score, WS, is calculated:

$$0.35 \cdot PH + 0.3 \cdot DB + 0.15 \cdot LoCH + 0.1 \cdot ToC + 0.1 \cdot RCS.$$

FICO 8 then applies a proprietary function $f : [0, 1] \to [350, 850]$ to WS to obtain the final FICO 8 score. This score is supposed to be positively correlated with the probability of non-default. The computation of WS only involves addition and multiplication so could be anonymized using a homomorphic encryption scheme such as the Paillier cryptosystem. The overall structure of this score can be thought of as a single perceptron computation of its inputs, so it is not surprising that neural net based attempts at computing scores anonymously such as Aldolfo, et al. [5] have been considered. We single out the Paillier cryptosystem because it is relatively simple compared to more general homomorphic cryptosystems such as those derived from Gentry [6], hence, the operations computed on the encrypted data will tend to be cheaper in terms of gas. Operations on the blockchain are generally visible to everyone, but if a credit score is initially encrypted, and the operations used to update it can be on the encrypted data, then we can ensure the borrower, or potential borrower, has some modicum of privacy if they use our system.

B. Decentralized finance (DeFi) overview

Decentralized finance (DeFi) products can be built on the Ethereum platform. Examples of products include decentralized lending, borrowing, trading, insurance, payment, exchanges, tokenized physical assets, etc. These are implemented on Ethereum as decentralized applications and smart contracts. A decentralized autonomous organization (DAO) is a crypto credit organization that has a flat hierarchy rather than tree like hierarchy; every participant has a stake and no single participant controls the organization in the conventional way. DAOs use smart contracts, with participants using governance tokens to vote on topics such as fund allocation.

MakerDAO[2] is a DAO that provides loans at predetermined interest rates. The basic steps of the loaning process are that the user deposits ether as collateral into a Maker smart contract, and the protocol lets one borrow/mint from MakerDAO a coin called DAI. Borrowing creates a Collateralized Debt Position (CDP) as DAI itself is a stablecoin pegged to the US dollar. Compound[3] is another leader in DeFi. Compound introduced their own blockchain – Compound Chain. Aave (formerly ETHLend)[4] is the DAO with the largest asset diversity. It also has lower collateral requirements as compared to MakerDAO. Aave offers products such as "DeFi blue chips" and "flash loans", which are trustless, and uncollateralized loans where both the borrowing and the repayment must happen in the same Ethereum block. Cred Protocol [5] is a start up working on providing decentralized credit score to the Aave protocol.

Some application require from users identity-related information. Verifying identity-related information typically requires a trusted third party that vouches for the correctness of the information and is done off-chain. One solution that is proposed is creating anonymous credentials. These are implemented with the use of zero-knowledge proofs that makes it possible to create credential verification without revealing sensitive identity attributes of the user to the verifier [7], [8].

[2]https://makerdao.com/
[3]https://compound.finance/
[4]https://aave.com/
[5]https://www.credprotocol.com/

Similar to our approach for the borrower registration mechanism are digital identities and verifiable credentials discussed in [9]. A digital identity is a digital way to identify a person. This person can prove that this digital identity belongs to them with the use of verifiable credentials.

Microsoft Azure provides a decentralized approach[6] for trust between banks. They propose the deployment of an efficient private Ethereum PoA (Proof of authority) blockchain, where member banks can establish their own nodes on a distributed ledger technology, where the credit score information of users are kept.

Another system that is related to our borrower registration mechanism is Hyperledger Indy [10], which is a decentralized credential management system where these credentials are realized based on Camenisch Lysyanskaya (CL) signatures [11]. Hyperledger Indy contains various tools that help with creating digital identities on blockchains or other distributed ledgers and they are interoperable across different administrative domains, applications, and any other silos.

III. Autonomous Lending Organization on Ethereum (ALOE)

In the ALOE system, a client's anonymity is maintained, but their credit score is publicly associated with their Ethereum address. The process begins with a *registration* step, which assigns an initial credit score to an Ethereum address based on the client's real-world credit score. Once a new loan has been created, borrowers and lenders connect to it to either request or invest funds. When the amount of the loan is met with sufficient demand from borrowers and with sufficient investment from lenders, the terms of the loan begin. The borrowers can then get their requested ether from the loan smart contract and later repay it. Should the borrower default on the loan, their real-world identity may be publicly revealed, hence allowing the lenders to pursue them for debt collection. The code for our implementation is being developed at https://github.com/taustin/cryptoCreditBureau/.

A. Borrower Registration

In the registration phase, the client shares their real-world identity in the form of their social security number (SSN) with the credit bureau. Registration involves the following entities:

- A *borrower* is a real person that has a unique SSN and an Ethereum address.
- A *notary* is a trusted real-world entity that validates the user's identity, credit score, and Ethereum address. It then divides the user's identity among various *auditors*, and invokes the *credit bureau smart contract* to initialize a credit score for the borrower. This initial score consists of the saved real world score together with a point in a credit space that we will describe in the next section. We trust that the notary does not store the association between the borrower and their Ethereum address, except for the secret shares sent to the auditors.
- The *auditors* are responsible for storing shares of the real-world identity associated with a borrower. We trust that they do not release their share of the secret, except as specified by the protocol. For simplicity, we assume that the set of auditors is small, fixed, and publicly known. With additional infrastructure, these assumptions could be relaxed.
- The *Credit Bureau Smart Contract* (CBSC) stores a mapping between Ethereum addresses and associated credit scoring information. This additional information consists of the credit score associated with the hash identity, a timestamp of when this association was made, as well as a point in our credit scoring space. The CBSC is responsible for connecting lenders and borrowers, for tracking loans and repayments, and for updating the borrower's credit score. The notary is the only one able to invoke certain methods of this smart contract.

For the registration phase, significant trust is put on the notary. If they fail to report accurate scores, a borrower's address may have a better credit score than it deserves. Likewise, if the notary records the mapping between the borrower's SSN and their Ethereum address, the borrower's anonymity is not protected. Since the notary's service is likely to be performed manually, it is not possible to enforce this forgetting. However, we note that if a notary performs their duties properly, they cannot be compelled to reveal the borrower's Ethereum address after the fact.

In our system, there can be multiple notaries operating independently. We further assume that the notaries are selected and paid by the borrowers. As a result, market conditions help to ensure that notaries perform their role honestly, since a notary with a bad reputation is unlikely to receive new business.

With this design, the auditors can collectively tie a user's Ethereum address to their real-world identity (as represented by the user's SSN). However, no individual auditor can do so.

We have implemented the CBSC as an Ethereum smart contract, called `CreditBureau`. CreditBureau supports the following functions relevant to registration:

- **`construct()`** creates an Ethereum account for a credit bureau. The creator of the `CreditBureau` (identified by `msg.sender`) also acts as the notary for verifying borrowers. The notary is the only person allowed to invoke the method `initScoreLedger`.
- **`verifyUnusedAddress(borrower_account)`** verifies that an address has not previously been used. This method prevents a borrower from resetting their credit score by revisiting the notary.
- **`initScoreLedger(borrower_account, ficoScore, timestamp)`** sets up the scoring for 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 calls `verifyUnusedAddress(borrower_account)` to ensure that the address has not been used previously.

[6]https://docs.microsoft.com/en-us/azure/architecture/example-scenario/apps/decentralized-trust

Figure 1 shows the registration process, detailed below: The borrower (1) sends their SSN and their Ethereum address to the notary who (2) verifies all real-world information. The notary then (3) sends the Ethereum address of the borrower to the CBSC. The CBSC verifies that it does not already have a credit score associated with the specified address. The notary then (4) divides up the SSN using a secret sharing scheme and sends it to the auditors. These messages should include the client's Ethereum address, the share of their SSN, the hash of the SSN (for verifying results), and the signature of the hash signed by the notary. This message is (5) repeated for all additional auditors. The notary (6) calls `initScoreLedger` with the borrower's address, their credit score, and a timestamp to register the borrower with an initial score. The notary should now discard all real-world information about the borrower.

B. Creating a loan

Creating a new loan must be initiated by an EOA, who is responsible for specifying the terms of the loan. We refer to this EOA as the *loan creator*. Since this operation requires ether from the loan creator, they may claim a fee from the interest of ether paid by the borrowers. Should there be any additional ether due to a lender claiming their fund early, the additional interest also goes to the loan creator.

C. Borrowing and Lending

The CBSC is responsible for connecting lenders and borrowers together by creating a new `Loan` smart contract. A *lender* is assumed to have an Ethereum account with some amount of ether. We add the following functions to our `CreditBureau` smart contract:

- **`createLoan(uint totalAmount, uint intRatePerMil, uint numPayments, uint secondsBetweenPayments, uint minCreditScore, uint fund_date) public returns (Loan)`** creates a new loan that borrowers and lenders can associate with. If the number of lenders and borrowers is not met by the `fund_date` of the loan, this money can be reclaimed by the lender. In our system, lenders are able to make withdrawals from loans after investing based on the reserve value.
- **`findLoan(uint amountNeeded)`** finds all loans registered to the credit bureau that could lend the amount needed given the user's requirements.

Our **Loan** smart contract supports the following contracts:

- **`invest()`** withdraws the amount `msg.value` from the lender (`msg.sender`), adding the balance to the loan smart contract. If sufficient funds have been received, then the time of this transaction is used as the starting time of the loan. This timestamp is used to calculate interest accrued.
- **`borrow(uint amount)`** is used by the borrower (`msg.sender`) to request a loan; The borrower's credit score and the amount requested must meet the loan requirements. This method involves a call to `getScore` which we will describe below and which might cost gas. Similarly, the `findLoan` operation above cost gas to compute and may have already computed the same `getScore`. To avoid expensive recomputations, these calls may be cached.
- **`isReady()`** returns whether or not the loan has funded.
- **`get$$$()`** transfers the amount of ether previously requested by the borrower once the loan has funded.
- **`makePayment(uint amount)`** is used by the borrower to make a payment towards a loan. This method succeeds if `msg.sender` has the necessary funds. As part of its implementation, this methods also changes the borrower's credit score.
- **`withdraw(uint amount)`** allows the investor to withdraw money from the account up to the invested amount plus interest. However, the lender cannot withdraw until after the loan funds; they may also use `withdraw` to reclaim their funds if the loan fails to either get enough lenders or borrowers.

In this section, we review the process of a borrower taking out a loan and later repaying it. We assume that the loan has been created already, but that it has neither been fully funded or gathered the full amount of borrowers needed.

To track the credit scores, we update the `CreditBureau` smart contract with the following functions:

- **`updateScoreBorrow(uint amount)`** tells the credit bureau that a loan has been made of the given amount. When a borrower receives a loan, the loan will in turn invoke this method on the CBSC. The transaction originator can then be used by this method to determine whose scores should be updated.
- **`updateScoreRepayment(uint amount)`** tells the CBSC that a loan has received a payback payment of `amount`. When a borrower makes a payment to a loan, the loan will in turn invoke this method on the CBSC. The transaction originator (`tx.origin`) can then be used by this method to determine whose scores should be updated. We currently do not support an EOA paying off the loan of another EOA, though that support could be added with additional complexity.
- **getScore(address client, uint amtRequested, uint intRatePerMil, uint numPayments, uint secondsBetweenPayments)** returns `client`'s credit score given the parameters on the amount requested. Details on the calculations for scores are discussed in Section IV.

Figure 2 shows a sequence diagram illustrating this process. The borrower (1) writes a transaction invoking the `Loan` smart contract (LoanSC) and requesting a loan. LoanSC (2) calls the CBSC, which (3) responds with the credit score of the borrower. LoanSC then (4) verifies that the borrower meets the loan requirements; if so, it adds the borrower to the list of borrowers for this loan and (5) updates the borrower's score with the CBSC. We note that the borrower's score is updated at this point as if they had already taken the loan.

After the above transaction is complete, the borrower must wait until the loan is ready, meaning that there must be

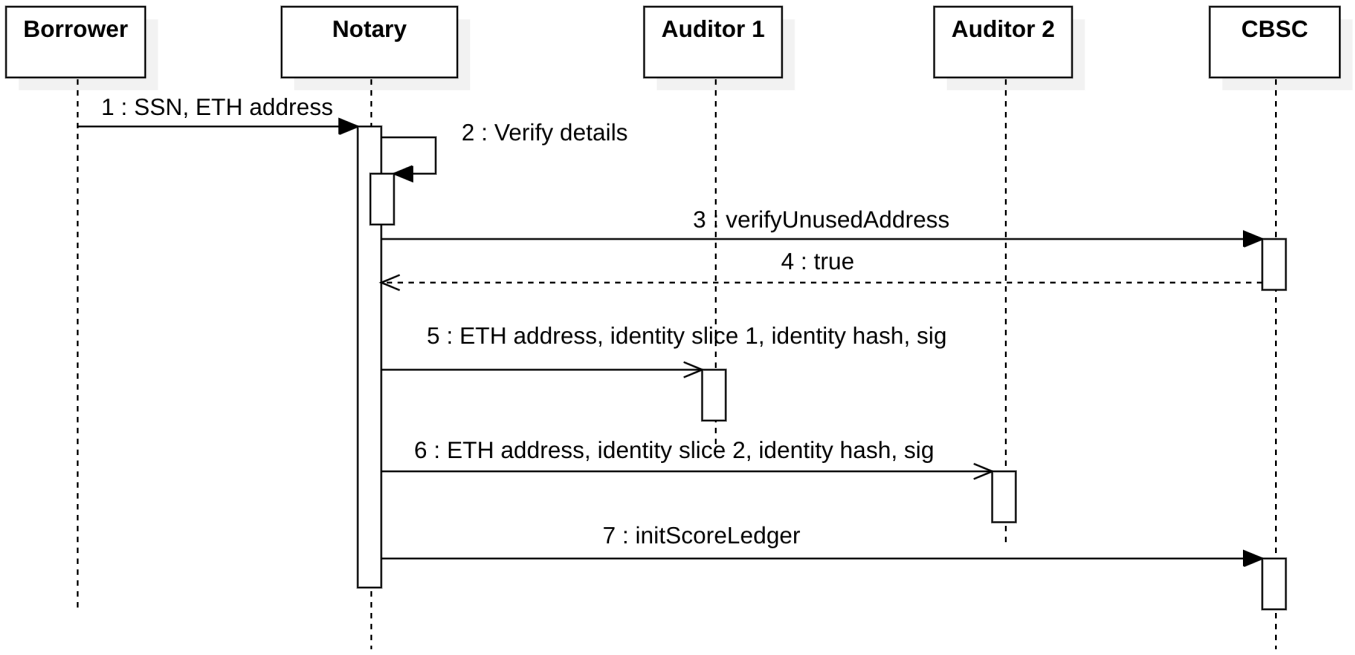

Fig. 1. Client Registration

sufficient lenders and borrowers. Borrowers will repeat steps 1-5. Lenders will instead (step 8) call the `invest` function. To determine whether the funds are available yet, the borrower may call the `isReady` function.

In steps 6-7 show the process the loan is not yet ready. However, after the final lender has invested (step 8), the client may call the `get$$$` function and receive their tokens (steps 11-12). When the borrower is ready to repay their loan, either in part or in whole, they call the `makePayment` function. LoanSC then contacts CBSC, updating the borrower's credit score accordingly.

D. Loan Default

If a client fails to repay a loan the lender may contact the auditors to reveal the client's identity. We assume that an external automated process periodically checks for loan defaults, and then writes a transaction whenever a loan default is detected. (By Ethereum's design, a smart contract can only be triggered by a transaction.)

The steps are as follows: A transaction (1) is written to the CBSC notifying it of the loan default. The CBSC (2) verifies the default and contacts the auditors with proof of the loan defaults. Each auditor (3) verifies the proof and responds with their share of the user's social security number (or other real-world identification). Any lender involved in the defaulted loan may then (4) call the CBSC to get the shares of the user's social security number. Off chain, the lender (5) combines the identity slices to reveal the identity of the client.

We note that anyone who could view the blockchain could also see the identity of the defaulted user, regardless of the permissions of the function to get the shares. If a user defaults on a loan, their real-world identity is essentially revealed to all users.

Once the borrower's real-world identity is revealed, lenders may pursue the borrower in more traditional manners to recoup their losses.

IV. Credit Score formulas

We now describe our credit scoring mechanism and how the function `getScore` is computed. Our mechanism is based on the k-nearest neighbors algorithm and is carried out on-chain. We deliberately chose that our system was to be computed on chain. This design allows the operations involved in the credit scoring to be publicly carried out and verified by the chain mechanism. If we perform the operations off-chain, we would need to verify on-chain that the score was updated honestly. While this might allow us to use a more sophisticated model, it introduces significant extra complexity.

Certain global properties of our model are computed periodically every fixed number of times `createLoan` is called. We imagine that `createLoan` is called less frequently than the borrowing, lending, withdrawing, or make payment operations. An individual calling the `createLoan` operation needs to pay the transaction fee so that there is sufficient gas to carry out these updates; on the other hand, as we detail elsewhere, the caller of `createLoan` is entitled to any leftover interest payments made by borrowers after all lenders have withdrawn their share along with an interest premium.

Our nearest neighbor model tracks five main ratios motivated by the component of FICO 8 as well as a set of window ratios. We store ratios in the smart contract using pairs of integers to avoid Ethereum dust. For each ratio, we also keep

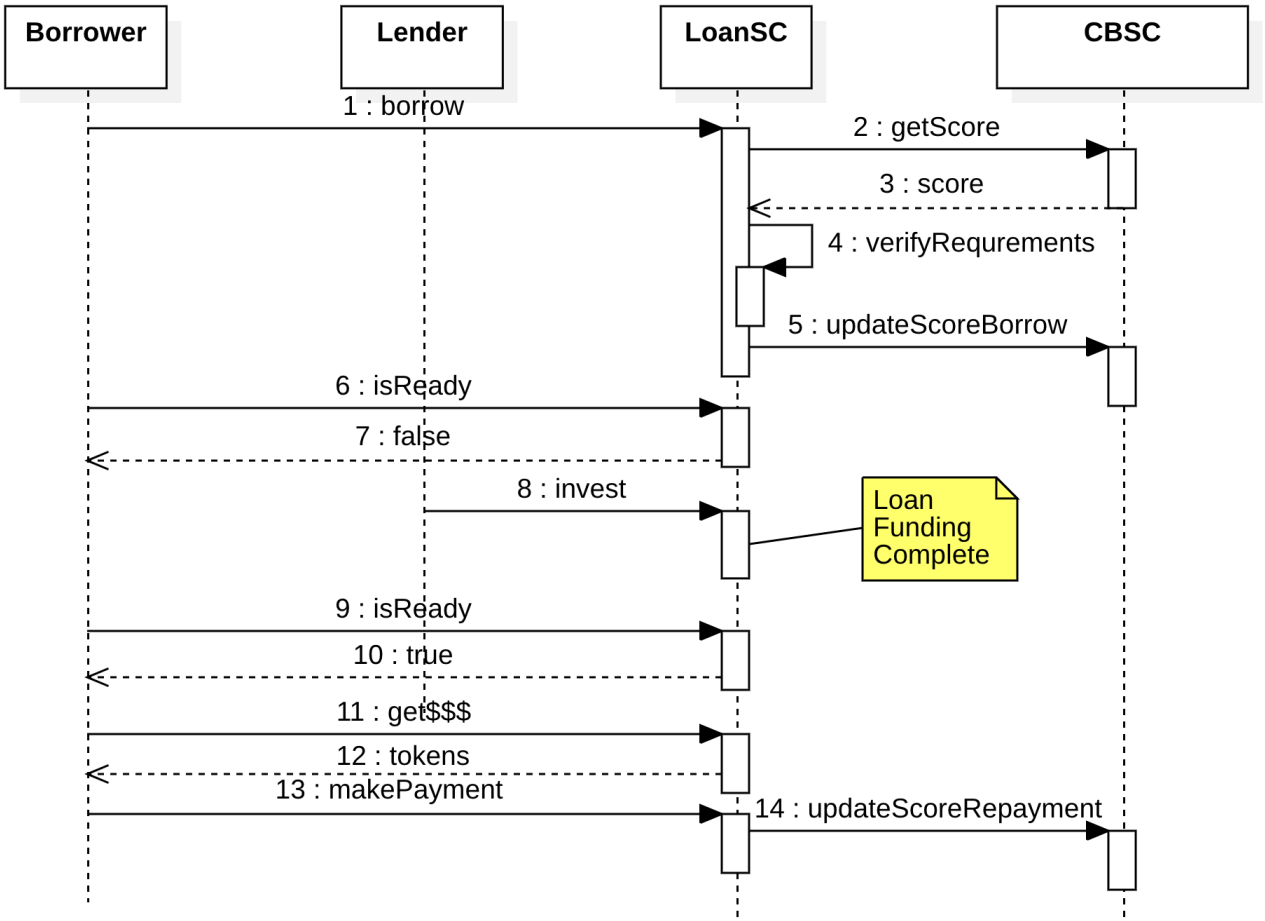

Fig. 2. Borrowing Process

track of the last time it was updated and other bookkeeping information needed to incrementally update it without having to store histories of all operations. Our nearest neighbor model consists of the following quantities:

- **Underpay Ratio (UPR)**. The ratio of time that the amount an individual owes is more than the amount that they should owe versus the total time span of all loans that that individual has ever carried. This component captures an individual's payment history as a single number. To compute this, the credit bureau needs to known the timestamp of the first loan of a borrower (FLT). Whenever a payment is made, it can also update based on the payment the total time the amount owed was greater than what should have been owed (TOT). This ratio is then $\frac{TOT}{NOW-FLT}$.
- **Current Debt Burden Ratio (CDBR)**. The ratio of current outstanding debt (COB) to the average outstanding debt (AOB) plus three "pseudo" standard deviations. The average outstanding debt can be computed as the total ever loaned (TEL) divided by TOT. Both COB, TEL, and AOB can be update on payments and new loans. To compute the standard deviation in AOB, we would need to compute a square root of sum of squares of the differences outstanding debts and the AOB. Each term in this sum would change any time that AOB changes. We would also like to avoid computing floating point operations like square root. So instead we compute a quantity we call Total Approximate Error (TAE). This starts out as 0 and when we update COB and AOB, we add to it the current value of $\Delta T \cdot |COB - AOB|$ where ΔT is the change in time since the last update. We then define a "pseudo"-standard deviation in AOB to be TAE/FLT. If CDBR is bigger than 1, we set its value to 1 when using it. This number captures the average debt burden of an individual.
- **Current Payment Burden Ratio (CPBR)**. The ratio of the current payment per day not to be behind on the loan versus the user's average payment per day not to be behind on the loans (since first loan was taken) plus three "pseudo" standard deviations. This quantity is computed using techniques similar to what we just described for CDBR. If this ratio is bigger than 1, we set its value to 1 when using it. This number captures on how big the current payment is compared to the payment the user has

traditionally had and captures to some degree the effect of different interest amounts.

- **Repayment Age Ratio (RAR)**. The loan size weighted age of all loans a user has had versus time since their first loan received. Let L_i and T_i denote the amount of the ith loan received by the user and the time at which this loan was made. Then this quantity is $1/(NOW - T_0) \cdot \sum_i L_i \cdot (NOW - T_i)$. It can be incrementally updated each time a loan is made, by updating TEL, and $WTEL := \sum_i L_i \cdot T_i$. From these quantities, we compute RAR as $1/(NOW - T_0) \cdot (NOW \cdot TEL - WTEL)$. This quantity is intended to correlate with length of credit history and recentness of debt.
- **Average Number of Credit Lines (ANCL)**. The quantity tracks the average number of loans the user has had since the user's first loan. Let S_i denote the number of simultaneous loans held when the ith loan was made. In the sequence of T_i of times of loans imagine we append one additional time T_{NOW}. We compute this as $1/(NOW - T_0) \cdot \sum_i^{NOW} S_i \cdot (T_{i+1} - T_i)$. This roughly tracks with a user maintaining a variety of kinds of credit.
- **Odds Stay Current w-day window (OSC-w)** The fraction of time that for the next window period, the user never owed more than they should have if regular payments were made. We compute OSC-w for 1 day, 2 days, 4 days, 8 days, 16 days, 32 days, 64 days, 128 days, 256 days, 512 days, 1024 days, 2048 days, 4096 days, 8192 days, and 16384 days (about 45 years).

Although we track these quantities, `getScore`'s value is not the weighted sum of these scores. We compute a user's score with the amount requested parameters from `createLoan`: `amtRequested`, `intRatePerMil`, `numPayments`, `secondsBetweenPayments`, and calculate what would be the user's values for UPR, CDBR, CPBR, RAR, NCLR if the loan were approved. We then make a vector (UPR, CDBR, CPBR, RAR, ANCL) and look up for the k nearest neighbors by Euclidean distance, their OSC-w's that bracket the loan period and take the inverse distance weighted average of these OSC-w's over the neighbors as the score. To allow a user who has no credit history, and hence no score vector, to obtain a loan, if the loan amount is below a threshold (for example, less than $1000 worth of ether), and the loan payment for the loan is less than a different threshold (for example, less than the equivalent of $100 per month), then the score returned by `getScore` is just the FICO score over 1000. So a FICO score of 850 in this case, would be returned by `getScore` as 0.850.

To solve the cold start problem, the quantities above can be calculated for loan data in other currencies than Ethereum. In particular, public domain data sets such as the national database model program, Kaggle, etc. can be used to populate an initial model for use in the CBSC. From this initial model, the value of k for the number of neighbors to consider can be computed by root mean square error estimation. These computations to create an initial model can be done off-chain. The ranges of each of our five quantities are between 0 and 1; during the training phase one can tune the accuracy of the starting and subsequent models by choosing to scale the size of individual components.

k-d trees are an obvious choice to store (UPR, CDBR, CPBR, RAR, ANCL) vectors as they allow for logarithmic lookup times of nearest neighbors. However, the use of such a structure would also entail a logarithmic overhead any time such a vector needs to be updated. This latter operation we would expect to occur with each loan payment, and hence, more frequently than `getScore` calls. For this reason, and because it is simpler to implement, especially on-chain, we chose the brute force implementation of k-nearest neighbors in which vectors are stored in an array indexed by hash of the user's address. So updating a vector for a user on a loan payment is an $O(1)$ operation. On the other hand, finding the k-nearest neighbors requires a linear scan of all our vectors.

The use of the brute force algorithm for nearest neighbors means that the cost to compute `getScore` increases in time and in gas as more loans are made. Currently, gas costs around 20-30 gwei [12] and the gas needed to compute the distance between two vectors and then compare to either insert or not insert into the priority queue of the current best k neighbors in typically in the low to medium hundreds, say 300. The cost to run a credit score in the real-world is usually less than say $50. Together these give a bound on the number of points to keep in our model for a comparable cost in the low tens of thousands. Using sampling, we size our initial model to meet these criteria. Thereafter, to maintain a limit on memory size we use two models for each loan duration window: an old and a new model. As new loan data arrive they are used to adjust the new model. A linearly weighted sum of the two models is used as the score based on the ratio of sizes of the two models. When the size of the new model reaches that of the old model, the old model is discarded and the new model becomes the old model. Finally a new, new model is started.

V. Conclusion and Future Work

ALOE allows for lenders and borrowers to connect in order to conduct loans. Loan credit scores are computed based on ongoing borrower behavior. Lenders are guaranteed a minimum score for the loans they invest in so they can choose the amount of risk they are comfortable with. Furthermore, we allow the identity of borrowers to be revealed should they default on a loan, while preserving the anonymity of well-behaving borrowers.

In future work, we plan to include more advanced forms of encryption. Proxy re-encryption [13] allows data to be encrypted on the cloud and re-encrypted for new users, with obvious applications for storage on a public blockchain. Homomorphic encryption [14] allows some operations to be carried out on data without decrypting it. Zhou and Wornell [15] provide a simpler homomorphic encryption scheme for integer vectors that would be implementable on chain.

An additional direction to explore would be to integrate an off-chain oracle for the k-nearest neighbor work, potentially

reducing the cost of the calculations significantly. However, providing the same security guarantees with this approach introduces some additional challenges.

Zero-knowledge proofs [16] could provide further anonymity so that only the minimum score is proved during borrowing. Additionally, our design assumes a single address per borrower. A borrower might want several addresses or to change an address over time. With additional infrastructure, we could tie these addresses to a single credit score. With these features, we hope to bring more traditional financial structures into the world of decentralized finance.

REFERENCES

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014.

[2] "Bitcoin: A peer-to-peer electronic cash system," 2009.

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

[4] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.

[5] L. Andolfo, L. Coppolino, S. D'Antonio, 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*, pp. 31–43, Springer, 2021.

[6] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, (New York, NY, USA), p. 169–178, Association for Computing Machinery, 2009.

[7] R. Muth, T. Galal, J. Heiss, and F. Tschorsch, "Towards smart contract-based verification of anonymous credentials," *Cryptology ePrint Archive*, 2022.

[8] 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.

[9] J. Sedlmeir, R. Smethurst, A. Rieger, and G. Fridgen, "Digital identities and verifiable credentials," *Business & Information Systems Engineering*, vol. 63, no. 5, pp. 603–613, 2021.

[10] T. Kuhrt, "Hyperledger indy." Hyperledger Foundation, 2022 (accessed August 12, 2022).

[11] J. Camenisch and A. Lysyanskaya, "A signature scheme with efficient protocols," in *International Conference on Security in Communication Networks*, pp. 268–289, Springer, 2002.

[12] R. deBest, "Average daily gas price of ethereum from august 2015 to may 16, 2022," 2022.

[13] S. S. D. Selvi, A. Paul, S. Dirisala, S. Basu, and C. P. Rangan, "Sharing of encrypted files in blockchain made simpler," in *Mathematical Research for Blockchain Economy*, pp. 45–60, Springer, 2020.

[14] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (Csur)*, vol. 51, no. 4, pp. 1–35, 2018.

[15] H. Zhou and G. Wornell, "Efficient homomorphic encryption on integer vectors and its applications," in *2014 Information Theory and Applications Workshop (ITA)*, pp. 1–9, 2014.

[16] Y. Han, H. Chen, Z. Qiu, L. Luo, and G. Qian, "A complete privacy-preserving credit score system using blockchain and zero knowledge proof," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 3629–3636, 2021.