New applications of blockchain technology to voting and lending

Mildred Chidinma Okoye

A Thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Information Systems Security) at

Concordia University

Montréal, Québec, Canada

November 2017

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mildred Chidinma Okoye**

Entitled: **New applications of blockchain technology to voting and lending**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________ Chair
Dr. W. Lucia

______________________ External Examiner
Dr. B. Caron

______________________ Examiner
Dr. A. Youssef

______________________ Supervisor
Dr. J. Clark

Approved by ______________________
R. Dssouli, Chair
Department of Concordia Institute for Information Systems Engineering

__________ 2017

____
A. Asif, Dean
Faculty of Engineering and Computer Science

Abstract

New applications of blockchain technology to voting and lending

Mildred Chidinma Okoye

With a lot of attention (even hype) given to blockchain technology, it is not startling that many researchers, developers, journalists, and start-ups have posited blockchain technology as the missing link for solving several problems with major interest within the financial sector.

Motivated by the current burst of interest around blockchains, we examine the feasibility of a few decentralized blockchain applications: specifically, voting and lending. The spiked interest in blockchain is the result of its perceived benefits, ranging from the removal of trusted parties and bureaucracy, lower costs, transparency, among others. In this thesis, we explore how the blockchain mechanism works, how it can be integrated with the concept of smart contracts (scripts that exist on the blockchain enabling the automation of processes), and how to create applications that operate on this decentralized platform.

We start by considering the viability of voting on a blockchain. Then, the bulk of this thesis focuses on a lending infrastructure for cryptocurrencies deployed on the Ethereum Virtual Machine. We present a cognitive walk-through for the deployment of a loan (peer to peer lending and bonds) on the blockchain by utilizing the transparency offered. A performance evaluation is given in terms of transparency, cost, security and reliability of the system. Limitations encountered as well as future work are discussed in the later part of this thesis.

Acknowledgments

This thesis embodies not only my effort but that of great people who supported me throughout this journey. First and foremost, I wish to express my gratitude to my supervisor, professor Jeremy Clark, assistance professor at Concordia University and an expert at Blockchain technology. He has been supportive from the start of this project, from coming up with a thesis topic to guiding me for over a year of development. I remember he often says "If it is not difficult, then it is not a thesis work" to encourage me during challenging periods. Dr. Clark has supported me not only academically but also financially through grants and moral support through to the completion of this thesis.

I would like to also express my thanks to the great co-authors that accompanied in the publication of the Blockchain voting paper, Yomna Nasser, Jeremy Clark and Peter Y A Ryan.

Lastly, I would like to thank my parents Mr. and Mrs. Emmanuel Okoye, my brothers and close friends for their undying and unfailing support throughout the course of this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital banking relies almost exclusively on financial institutions serving as trusted middle-men to process electronic transactions. Although the scheme is adequate for most transactions, it suffers greatly from the integral weaknesses of the internet — the model is based on trust. Though it is possible, it is very difficult to make transactions non-reversible since financial institutions play the role of mediating disputes to satisfy their customers. This increases the costs of transacting as mediation is expensive and needs to be factored in. It becomes less practical for small casual transactions to be digitized, and limits are imposed: e.g., on the minimum transaction size per day. The possibility that transactions can be reversed also leads to mistrust. In fear, merchants are left with little choice but to hassle customers for more information than they would otherwise need. Only a small percentage of fraud is accepted as unavoidable. By using physical currency, these costs and payment uncertainties can be avoided, but no mechanism existed that allowed payments over the internet without a trusted party until Bitcoin's introduction in 2008 [32].

Bitcoin, the very first decentralized cryptocurrency, has emerged as the most widely-deployed, virtual global currency, and has awakened many copycat currencies. Not surprisingly, Bitcoin has gained much attention from the financial, technology, and academic sectors. The core technical innovation fueling these schemes is the Nakamoto consensus protocol, which sustains a distributed ledger widely referred to as a blockchain. The distributed nature of the blockchain enables trustless networks to agree on transactions, therefore eliminating the need for an intermediary, which in turn boosts the reconciliation period between transacting parties. The substantial use of cryptography, a vital characteristic of blockchain networks, puts verifiability and transparency behind the meaningful interactions on the network.

Currently one of the hottest areas within cryptocurrencies, smart contracts are agreements between two or more parties that can be automatically enforced without the need for an intermediary. Today's leading smart-contract platform is called Ethereum, whose blockchain stores long-lived programs, called contracts, and their associated state, which includes both data and currency (ether). These programs are immutable just as data on the blockchain is, and users interact with them with a guarantee that the program will execute exactly as specified [3]. Combining these concepts allow for a distributed and heavily automated workflow [11].

In this thesis, we take two current applications that are highly trust-centric — voting systems and lending — and we explore how the transparency characteristic of the blockchain can enable a more transparent and automated application that minimizes the role of trusted third parties.

1.1 Contributions & Organziation

First we contribute an evaluation framework of different distributed ledger technologies, where we present a taxonomy and criteria that is used for comparative analysis. This is given alongside the preliminaries of Bitcoin, blockchains, and Ethereum in Chapter 2.

The next chapter (chapter 3) presents a systematized existing knowledge on blockchain technology and verifiable voting technology, showcasing what the landscape looks like for their composition and commenting on future work, informed by past work. The goal here is to provide a justified research agenda by bringing into focus where there is potential, and pruning out what seem to be dead-ends.

Finally, the main aim of this project is to introduce some applications of blockchain technology that go beyond the creation of new cryptocurrencies. This is actualized by developing a decentralized lending infrastructure system that harnesses the transparency of the blockchain to provide a system that is translucent and requires less trust on third party. This part of the project is implemented on the Ethereum Virtual Machine via smart contract using Solidity Language. We present a cognitive walk-through for the deployment of a loan on the blockchain by utilizing the transparency offered by the blockchain technology. This chapter also contains the challenges encountered during the deployment stages.

An evaluation performance in terms of transparency, cost, security and reliability of the system was conducted and limitations encountered as well as future work are discussed in the later part of the thesis.

Chapter 2

Preliminaries

Motivated by the current burst of interest around blockchains, we examine the feasibility of applications such as voting and a lending infrastructure on the blockchain. The spiked interest in blockchain is because of the benefits it offers ranging from the removal of trusted parties, lowered bureaucracy, lowered costs, transparency, and many more. In this thesis, we explore how these mechanisms work, as well as integrating the concept of smart contracts (scripts that exist on the blockchain that allow for the automation of processes) to develop an application that operate on a decentralized platform. In this section, we describe Bitcoin in sufficient detail to understand all contribution made in this thesis. Although not necessary for understanding this thesis, additional information on Bitcoin can be found in [32].

The motives behind why majority of the people trust the Bitcoin system are peculiar to each individual perspective and cannot be fully explained. However, from a broader perspective, ample of this trust emerges from the known fact that Bitcoins historical transaction records, the security control implemented and the software used are virtually open for everyone for verdict which is in part contrary to most traditional organizational system like the banks (who are becoming less trustworthy). This term generally, referred to as

Transparency seems to be valuable to users in almost every facet of life. Also another reason why many trust the Bitcoin also lies in the fact that its being managed by the users of the system rather than a government which in turn gives the users more control over their money. It also affords one the option of performing secure transactions across the world rather than using wire transfer. A lot of merchants frown against the ability to reverse transactions and this constitute to another reason why users trust the system. The security controls implemented prevents reversal of transactions when confirmed and also against the risk of fraud associated with credit cards. This also applies to other cryptocurrencies like Ether (Ethereum) that uses the same Nakamotos consensus algorithm.

2.1 An Overview of Bitcoin

While there is some debate about which economic system emerged first historically, it is still informative to consider three basic systems in comparison to each other: barter, credit, and cash. In a barter system, goods trade amongst participants simultaneously without the need for trust (other than in the good itself) but coordinating trades in time and space is problematic, in particular when trades involve more than two people. Credit allows trades to be made without receiving anything immediate in return, but requires counter-party trust. With cash, cash is received in return for goods which resolves the 'coincidence of wants' issue or barter and minimizes the counterparty risk of credit (cash still involves some trust: that the cash is legitimate, that it will preserve its value over time, and that others will accept it in the future). Cash also enable anonymous payments. Any payment in cash involves no bank for verification and the merchant is of no obligation to know who the user is. On the other hand, credit cards allow for online payments but have two disadvantages. For one, it offers no anonymity as a credit card is issued with a name and transactions can be traced backed to each user. Second, it enforces trust in third parties as well as high fee due to the bureaucracy contained within the system. Hence, the need for a form of online payment that offers the same benefit as cash arose [33].

2.1.1 History of Cryptocurrency

The study of electronic cash began with David Chaum, an American cryptographer. The desire for privacy and a trustless system for online commerce led to the idea of a tokenized representation of physical coins and paper notes. Thus, in 1983, David Chaum introduced blind signatures [8] which enabled Alice to be issued a coin by a bank with a serial number that Alice could change before transferring it to Bob. The bank would not recognize the new coin however the bank's signature endorsing the value of the coin would be kept intact. These properties are a product of mathematical operations within the RSA signature scheme (and have since been adapted to other signature algorithms). Chaum's company DigiCash sprang up as a result and the invention of this blinded caused extraordinary attention from the press and industry. Unfortunately, the company failed to secure a critical mass of banks to support the product and the company filed for bankruptcy in 1998.

In 1998, Wei Dai's introduced b-money [1], the idea of creating money by solving computational puzzles, as well as implementing a decentralized consensus. However, the proposal was limited on details; a concrete implementation of the decentralized consensus was hand-waved away. In 2005, Hal Finney introduced a concept of "reusable proofs of work", a system which uses ideas from b-money together with Adam Back's computationally difficult Hashcash puzzles to create a concept for a cryptocurrency, but once again fell short of the ideal by relying on trusted computing as a backend [7].

[1] url//www.weidai.com/bmoney.txt

2.1.2 Basic Cryptography

The best known cryptographic function is encryption, which is used to hide messages. Bitcoin does not use encryption however it uses two other cryptographic primitives: hash functions, used to uniquely fingerprint data, and digital signature algorithms used to endorse data with a verification key. Functions in cryptography generally either accept an input only (hashes) or an input and a key (signatures) to generate an output.

Hash Functions

A hash function take an arbitrary-sized input and deterministically generates a fixed-sized output that serves as a binding fingerprint of the input. Generally, hashes can be consider a one-way function: the *pre-image resistance* property of a hash function states that it is computationally infeasible, given an arbitrary hash output y, to find any input x such that $y = \mathcal{H}(x)$. It should also be infeasible to find a *collision*: two different values x_1 and $x_2 \neq x_1$ such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. A weaker form of collision resistance states the infeasibility of finding $x_2 \neq x_1$ given x_1 (as opposed to being able to choose x_1) such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. The word weak is from the perspective of the designer of the hash function: insecure hash functions might achieve this property while admitting collisions. In Bitcoin, SHA256 and RIPEMD-160 are used. The former has all three properties, while RIPEMD-160's collision resistant is well below the NIST standard (which requires at least 224-bit output to acheive a security level of 112 bits). It is speculated that this is not a problem in Bitcoin, with some debate. In Bitcoin, SHA256 is used as a building block for the proof of work (PoW), hash trees, and hash chains. The composition of SHA256 and RIPEMD-160 are used to generate a Bitcoin address from a public key for a digital signature scheme.

Digital Signatures

A digital signature scheme allows an arbitrary-sized input to be signed such that it cannot be modified by anyone other than the signing party. Signatures utilize two keys: a secret private key known only to the signer used to generate the signature, and a mathematically related publicly available key used by anyone to verify that the signature was generated with the associated private key (without revealing the value of the private key). The Digital Signature Algorithm (DSA) is a signature based on the computational intractability of discrete logarithm problem (DLP). DSA works in any mathematical group with a DLP-style intractability problem. Originally it was proposed in prime-ordered groups of positive integers, but later, more efficient and compact groups based on elliptic curves over finite fields were found and analyzed. Bitcoin utlizes the elliptic curve version of DSA, called ECDSA. The advantage of ECDSA over DSA is faster computations, smaller keys, and smaller signatures [24]. In Bitcoin, ECDSA is implemented on the secp256k1 curve.

2.1.3 Hash chains and trees

In Bitcoin, hash trees or merkle trees [31] are essential for ensuring data integrity. Each transaction created by nodes within the network are grouped into blocks. These transactions are first hashed individually and then the resulting hash values are then hashed in pairs (forming a binary tree) until a single hash value is derived. This is referred to as the Merkle root and it serves as a binding commitment to every element in the tree (which can be as big as one wants). Hence each block has a single hash value that represent all transactions contained within the block. This structure allows for the detection of any changes within a block of transactions as a change of even a single bit in any transaction would result in a completely different Merkle root value.

The purpose of the Merkle tree is also to allow for easy retrieval and verification of branches within the blockchain. This is particularly useful for thin clients (like mobile

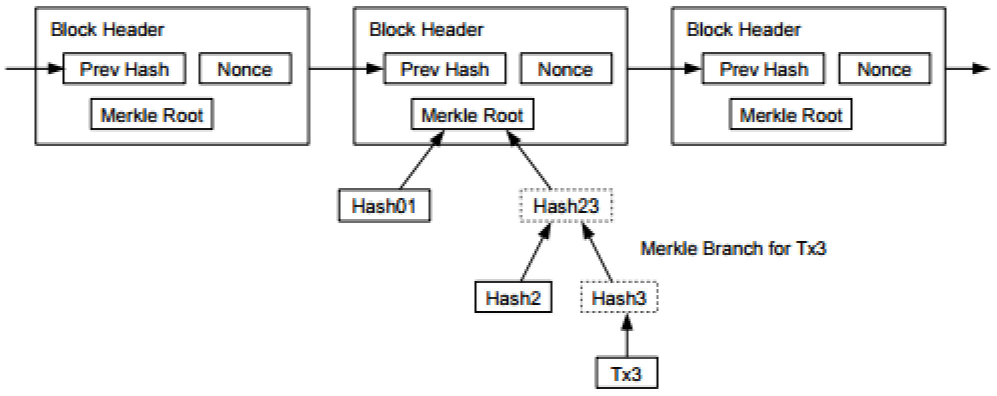


Figure 2.1: The bitcoin Merkle tree and adjoining Hash Chain. [32]

phones) which use a protocol called Simple Payment Verification, where a semi-trusted server forwards the Merkle branches of transactions that match the client's keys/addresses. This allows the thin client to do some verification without downloading the full blockchain (which at the time of writing is approaching 100 GB). Since in a Merkle tree, hashes propagate upward to the merkle root, a malicious server cannot insert or modify a transaction without changing the Merkle root. An incorrect Merkle root will be ignored by the network and not incorporated into the set of transactions.

Each block is represented by a hash value (the hash of the block header that contains an asserted timestamp, the Merkle root of the transactions in the block, and the previous block hash) that is used as an input to a puzzle to be solved for the next new block (the block header also contains a nonce value used for the purpose of this puzzle, described next). With this format of linking the current block to the previous block, all blocks are chained together thereby ensuring the last block is a binding commitment to the entire past history of the system. This concept is referred to as Hashchain introduced by Haber and Stornetta [22].

2.1.4 Proof of Work

When Alice receives a payment from Bob, she needs to ascertain that Bob did not also send that same money to Eve—in other words she need to be assured that Bob is not double spending the money. She could be certain if there is a central authority that keeps record of all transactions. Then she could verify the state of any transaction she receives. In order to prevent a double spend attack in a decentralized system, a consensus mechanism is needed and this becomes a tricky matter where any node can join or leave. Bitcoin uses a computational puzzle, called a proof of work, to ensure consensus can be formed if the majority of computational power on the network belongs to honest nodes.

The concept used in the Proof of Work can be traced down to Dwork and Naor [17]. They proposed a mechanism for combating spam where the sender would be required to compute a solution to a moderately hard puzzle (something that under normal conditions would take a computer several seconds or minutes) before sending an email. The verification process is however trivial for the recipient taking only millisecond or microseconds. Messages without a correct solution are ignored. Hashcash [2] implemented this idea in such a way that the puzzle involved computing the hash of the email (including its metadata) a large number of times. Each time this computation is performed, a counter value (called the nonce) included in the hash input changes, and the computation ends when the output of the hash function happens to have a specified number of leading zeros. Hence the birth of the proof of work concept used in Bitcoin and Ethereum.

As stated earlier, one major problem with previous existing digital currency was decentralization (how to make a group of unknown people within a network reach an agreement on the state of the network). To solve the consensus problem in a decentralized setting, controlled and censored by no authority, Bitcoin adopted a novel Byzantine consensus protocol called the Proof of Work consensus mechanism (or Nakamoto consensus) [34]. The main condition of Nakamoto consensus is that 51% of the computing power must be controlled by honest nodes in the network. Because computational power is difficult and expensive to acquire, it is almost impractical for a single entity to control a majority of the power in the network, sidestepping Sybil attacks [16].

It's necessary to bear in mind that the sustainability of the network depends on reparamaterizing the difficulty of the proof of work. Miners search for a nonce that results in a block header hash output that is less than some target number, also referred to as difficulty. This number is recalculated (increased or decreased) every 2016 blocks (roughly every two weeks) to bring the average time between each block to 10 minutes. Nodes are incentivized to perform this computational operation (often called mining) by collecting newly minted coins and transaction fees.

2.1.5 Currency

In the Bitcoin blockchain, the notion of mining requires miners to solve a puzzle to add a block onto the end of the blockchain. The first miner to solve the problem will broadcast their winning block and pay themselves a mining reward, which as of the writing of this thesis is 12.5 BTC. This reward was 50 BTC in 2008 when bitcoin was invented and it halves every four years until it approaches zero. The total number of Bitcoin that will come into circulation is 21M BTC. The reward given to miners is a form of distributing the currency within the network (also known as bootstrapping) since there are no central authority to issue the currency. Miners also receive transaction fees from every transaction included into a block. The fee is a payment to the miner for verification performed on the transaction. The transaction fee is denoted as the difference between the input amount and the output amount of any transaction.

2.1.6 Scripting

Bitcoin's Script is a simple scripting language that maps human-readable assembly-style code known as ”opcodes” into operations. A script is included in every Bitcoin transaction and it describes who can spend the money in a future transaction. The vast majority of transactions follow a default script which provides the Bitcoin address of the spender and specifies that the spender must sign their transaction with the key that corresponds to this address. However other scripts are possible.

Scripts are made up of a sequence of instructions given in the transaction holding the funds (called an unspent transaction output) and the new transaction spending the money (called a transaction input). The scripts are concatenated and executed linearly on a stack with no branching (jumps, loops, etc). Hence, execution time is bounded above by the length of the script after the instruction pointer. This constraint prevents denial of service attacks on the nodes validating the blocks [41]. Since every node on the network has to validate every script ever run to synchronize to the latest block without any compensation for doing so, available opcodes are very judiciously chosen with a emphasis on things enabling currency-based applications. Bitcoin's scripting language is not Turning complete. Value-blindness prevents Unspent Transaction Outputs from providing fine-grained control over what amount of money can be withdrawn [7]. Unspent Transaction Outputs are either spent or unspent making it difficult to multi-stage option contracts that are complex. Finally, transactions cannot reference blockchain values, such as the nonce or previous hash, which might otherwise provide a good source of randomness for applications such as gambling.

2.2 An Overview of Ethereum

We now describe Ethereum, a decentralized blockchain-based system that uses a currency, called Ether, in a similar fashion to Bitcoin however also enables a verbose, Turning complete scripting language that allows for full fledge smart contracts and other non-currency applications that will be executed correctly according to the consensus of the decentralized network of nodes maintaining the state of all Ethereum contracts. Proposed by [7], Ethereum aims to ameliorate the notion of altcoins, on-chain meta-data and scripting thereby resulting in the creation of arbitrary consensus-based application that are standardized, inter-operable, feature complete and scalable. We simply a few details of Ethereum; for example, even though its blockchain structure is significantly different from Bitcoin's, it is inconsequential for the results of this thesis to think of Ethereum contracts running on a blockchain based on Bitcoin's design.

2.2.1 The Ethereum Virtual Machine

Ethereum much like Bitcoin is a shared transaction database often referred to as Blockchain. It operates in such a way that all participating entities (nodes) within the network can read entries (transactions) and make changes by creating transactions for which a consensus is needed. A transaction in its basic form describes the transfer of Ethereum currency (Ether) from one or more accounts (input addresses) to one or more accounts (output addresses). Ethereum addresses are derived from a digital signature scheme which are not centrally allocated or registered by a third party as these addresses become active during the first transaction.

As a decentralized network, consensus is reached by incentivizing node within the network with ability to create (mine) new Ether for every valid block created. Over time, these blocks form a linear sequence which are added roughly every 17 seconds which helps to maintain and update the blockchain. Further details of the Ethereum Consensus architecture are not needed in this section. However, it is useful to note that nodes in the network partake of the consensus protocol by downloading the blockchain and cryptographically verifying the integrity of the transactions contained therein.

The Ethereum Virtual Machine (EVM) is an isolated, sandboxed runtime environment for developing smart contracts in Ethereum. Codes deployed within the EVM have no access to other networks, processes or file system.

2.2.2 Ethereum Accounts

In Ethereum, there are two forms of accounts each sharing the same 20-byte address space: An External account, controlled by private-public keys pairs with no code contained therein and a contract account, controlled by the code stored within the account. Although both addresses are treated equally by the EVM, the address of the external account differs from the contract address in that as the former is determined by the public key, the latter is determined at the contract's creation time derived from the combination of the creator's address and a nonce that represent the number of transactions executed by the creator's address key pair.

Messages can be sent from an external account by generating and digitally signing a transaction. In other for a contract address to read and write to its internal storage, create other contract or send messages, its code needs to be activated by a message received either from an external or internal account. In general, an Ethereum account contain four fields:

- A balance in Ether (usually stored in 'Wei' by the EVM)
- A contract code (if empty, it represents a normal the account, otherwise it represents a contract account)

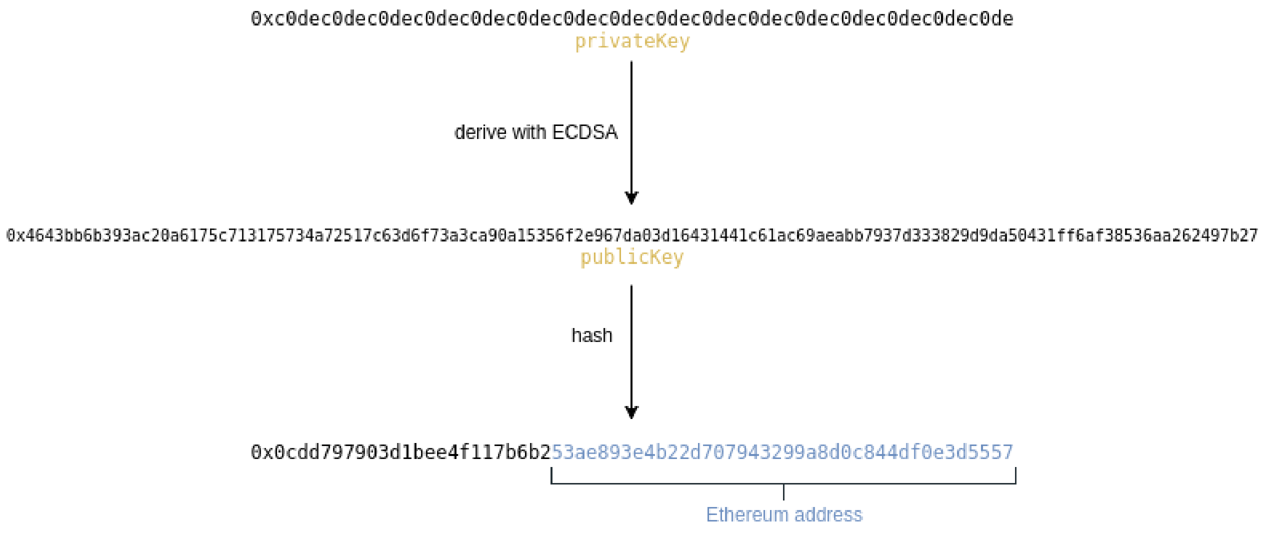

0xc0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0dec0de
privateKey
derive with ECDSA
0x4643bb6b393ac20a6175c713175734a72517c63d6f73a3ca90a15356f2e967da03d16431441c61ac69aeabb7937d333829d9da50431ff6af38536aa262497b27
publicKey
hash
0x0cdd797903d1bee4f117b6b253ae893e4b22d707943299a8d0c844df0e3d5557
Ethereum address

Figure 2.2: An Ethereum address derived from its public key. [13]

- A storageRoot which has a persistent key-value store mapping 256-bit words to 256-bit words (default set as empty)
- A nonce that represents a counter to ensure each transaction is processed only once.

An Ethereum address which represents an account is derived as follows:

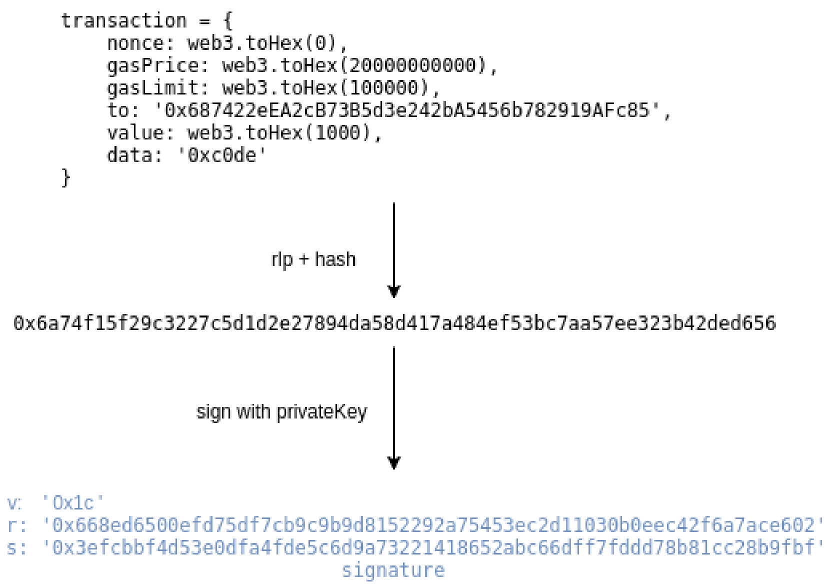
First, a Private Key, a randomly selected positive integer (represented as a 256-bit data or a byte array of length 32 in big-endian form) is generated [42]. Second, a Public Key, is derived from the private key generated using the Elliptic Curve Digital Signature Algorithm (ECDSA) with secp256k1s curve. Finally, the address is derived from the keccak-256 hash of the hexadecimal form of a public key where only the last 20 bytes are kept. It useful to note that this differs from the Sha-256 used by Bitcoin. An example of an Ethereum address usually denoted in hexadecimal format (base 16 notation) is “cd2a3d9f958e13cd977ec05abc7fe734df8cc826”. This is often indicated explicitly by adding 0x prefix to the address.

2.2.3 Transactions

In Ethereum, 'messages' are to some extent like 'transactions' in Bitcoin, but with three vital variances. First, an Ethereum message can be generated by an external account or a contract account, while a Bitcoin transaction can only be generated by an external account. Second, Ethereum messages have an explicit option to contain data. Finally, if it is a contract account, the recipient of an Ethereum message has the option to return a response; this means that messages in Ethereum also encompass the concept of functions. The term 'transaction' is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account [7]. The inputs to an Ethereum transactions or message are as follows:

- **The Nonce:** a counter for the number of outgoing transactions, starting with 0
- **The GasPrice:** the price in ether per unit of gas to pay to the miner per computational step. It defaults to a price of 20 GWei (0.000000020 Ether).[2]
- **The GasLimit:** the maximum gas allocated by the sender to be spent to process the transaction.
- **Value:** the amount of ether to be sent to the recipient account of the transaction, if empty, a contract is created
- **A Signature:** identifies a sender and provides an intention to send the message via the blockchain to the recipient
- **Data:** could be a function call to a contract, an arbitrary message or a code to create a contract.

[2]https://ethstats.net/



```
transaction = {
    nonce: web3.toHex(0),
    gasPrice: web3.toHex(20000000000),
    gasLimit: web3.toHex(100000),
    to: '0x687422eEA2cB73B5d3e242bA5456b782919AFc85',
    value: web3.toHex(1000),
    data: '0xc0de'
}
```

```
0x6a74f15f29c3227c5d1d2e27894da58d417a484ef53bc7aa57ee323b42ded656
```

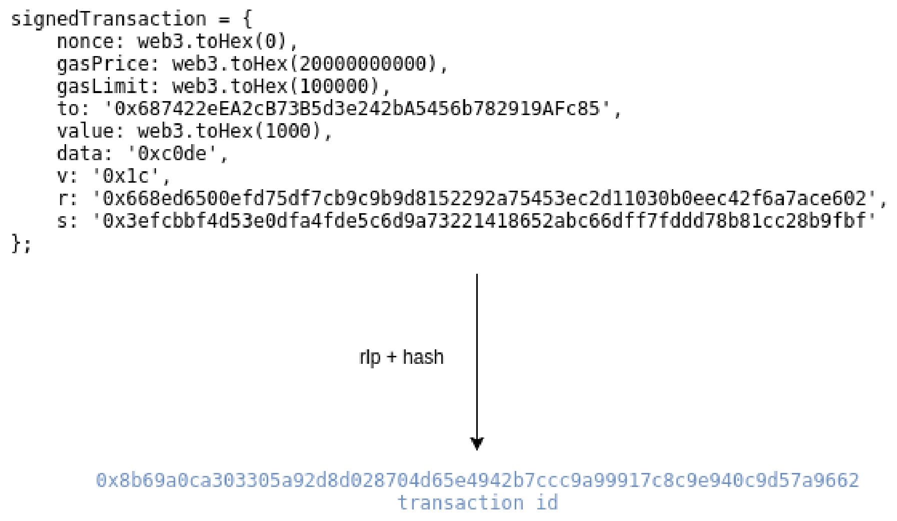
```
v:  '0x1c'
r: '0x668ed6500efd75df7cb9c9b9d8152292a75453ec2d11030b0eec42f6a7ace602'
s: '0x3efcbbf4d53e0dfa4fde5c6d9a73221418652abc66dff7fddd78b81cc28b9fbf'
                                signature
```

Figure 2.3: A transaction sending 1000 wei (1 ether $= 10^{18}$ wei) of ether with a 0xc0de message. [13]

Generally, there are three types transactions supported on Ethereum: *Transfer* of Ether from one party to another, *Creation* of a smart contract and *Transacting* with a smart contract. These transactions only differ in their inputs. With the gaslimit, nonce and gasprice present in all types of transaction, a Transfer of Ether from one party to another would only require a “to” and “value” input; a transaction to create of a smart contract would require a ‘value’ and a ‘data’ input; a transaction interacting with a smart contract would require ‘to,’ ‘value,’ and ‘data’ inputs.

2.2.4 Gas

When a transaction is created in Ethereum, a fee must be paid to the miner that executes the transaction and update the output of the transaction to the Ethereum Blockchain. This fee is measured in gas, where gas refers to the number of instructions needed to execute a transaction within the Ethereum Virtual Machine. Therefore, there are two parameters used



```
signedTransaction = {
    nonce: web3.toHex(0),
    gasPrice: web3.toHex(20000000000),
    gasLimit: web3.toHex(100000),
    to: '0x687422eEA2cB73B5d3e242bA5456b782919AFc85',
    value: web3.toHex(1000),
    data: '0xc0de',
    v: '0x1c',
    r: '0x668ed6500efd75df7cb9c9b9d8152292a75453ec2d11030b0eec42f6a7ace602',
    s: '0x3efcbbf4d53e0dfa4fde5c6d9a73221418652abc66dff7fddd78b81cc28b9fbf'
};
```

```
0x8b69a0ca303305a92d8d028704d65e4942b7ccc9a99917c8c9e940c9d57a9662
transaction id
```

Figure 2.4: The transaction ID generated [13]

to indicate how much Ether the sender is willing to spend for a transaction to be successful:

- **Gas price:** This is the amount Ether per unit of gas that started with default price of 9000 Wei (9 x 10^{-15} Ether) and now 20 GWei.

- **Gas limit:** To avoid exponential blowup and infinite loops in code, a limit is to set on the number of computational steps of code execution it can spawn for the transaction, including the initial message and any other messages produced during execution. There is an upper limit on the fees of a single transaction in an Ethereum block which confines this value typically to less than 1,500,000 which at the time of this writing has increased to 6,712,392.

These combined parameters decide the maximum amount of Ether to be spend on transaction and how quickly a transaction takes place is a factor of the gas price. If a transaction execution 'runs out of gas,' all changes to the state are revert — except for the fees paid. In the common case where a transaction execution halts with some unused gas, the rest of the fees is refunded to the sender. A user's client will test(execute) the transaction and compute the amount of gas required and set this as a default gas limit before sending the transaction to the Ethereum network. There are however a variety of reasons this estimate might be inaccurate: the most common is that the data structures the code invokes change between sending the transaction and its execution by the miner (e.g., by other simultaneous transfers to the same contract).

2.2.5 Memory, Storage and Stack

Codes in Ethereum contracts are run from a low-level, stack-based bytecode language, referred to as Ethereum virtual machine (EVM) code. The codes are series of bytes, where each byte represents an operation. Generally, code execution is an infinite loop that involves repeatedly executing operations at the current program counter that starts at zero and then incrementing the program counter by one, until the end of the code is reached, an error or STOP or RETURN instruction is detected, or the contract runs out of gas. These operations can store and access data in three different ways,

- Memory: which is linear, and can be addressed at byte level, while reads are restricted to a size of 256 bits, writes can be either 8 bits or 256 bits wide;
- The stack: a 32-byte valued last-in-first-out container can be pushed and popped;
- The storage: the contract's long-term key/value storage, a store where both keys and values are 32 bytes.

Storage persists for a long time as opposed to memory and stack which reset after computation ends. The value, sender and data of the incoming message, as well as block header data, can also be accessed by the code and the code can also return a byte array of data as an output.

Table 2.1: An evaluation framework for distributed ledger technologies.

CRITERIA / LEDGERS	Append only	Non Equivocate	Tranparency	Liveness	Anonymity	Permissionless	Carbon Dating	Avoid PoW	Privacy Preserving	Public Auditable	Decentralized	User store Entire log	Resilient to attack	Accountability
Bitcoin Blockchain	●	●	●	●	◉	●	●	⊘	⊘	●	●	◉	●	⊘
Certificate Transparency	●	●	●	●	⊘	●	⊘	●	⊘	●	◉	⊘	◉	●
Coniks	●	●	●	◉	⊘	◉	⊘	●	●	●	◉	⊘	◉	●
Naïve Transparency Overlay	●	●	●	●	◉	●	●	⊘	⊘	●	◉	⊘	◉	⊘
Rscoin	●	●	●	●	◉	◉	⊘	●	◉	●	◉	⊘	⊘	●

●	Has the full property
◉	Somewhat has this property
⊘	Does not posses such property

2.3 A Survey on Distributed Ledgers

A distributed database can be viewed as duplicates of database stored in different physical locations to ensure availability of information. This enables responses to be received with high certainity for every request made, and reliability even under conditions where some nodes fail. However, distributed databases do not address important security criterion: namely, security guarantees that users of the system see the same state at any given time (non-equivocation) and it cannot change (immutability). Hence, the introduction of distributed ledgers. A Distributed ledger is a ledger that is replicated among all the nodes in a peer to peer network without the need of a central party to coordinate actions or be authoritative over any aspect of the protocol. Each node's copy of the ledger is guaranteed to be identical using a consensus algorithm and the use of cryptographic signatures ensures the ledger is unforgeable.

The following is a set of desirable properties we may want from a distributed ledger. We use them to evaluate and classify several proposals from the literature. A summary of our evaluation is provided in Table 2.1. We provide the details of the evaluation in Appendix A.

CRITERIA

(1) Append-only: The logs are denoted by the leaves in a Merkel tree. Each level in the tree is a cryptographic hash of the nodes below where the root is at the top and the leaves are at the bottom following an odd fact that trees grow downward. The properties of a cryptographic hash ensure that each node is a summary of all the nodes below it thereby making the root of the tree a summary of all of the leaves. This property allows data to be added only to the end of a log as the logs are chained to each other in such a way that an insertion of any data in the middle of the log would be detected.

(2) Non-Equivocate or Consistency: A log should be consistent among different users and any attempt to present different views should be detected. Two users can effectively verify that they are both seeing the same tree by simply comparing the root hash.

(3) Transparency: All operations within the log should be visible to all users in real time. Transparency of the log does not necessarily mean that the published logs are accurate, but users can trust them knowing that the owners of the information (certificates or transactions) will promptly detect any modification, tampering or misbehavior.

(4) Liveness: An assurance that data sent to be entered into the log is guaranteed to be inserted in the log.

(5) Anonymity: No real world identity is tied to the identity obtained within the named system.

(6) Permission to Join: This property states if a permission would be required by user to join the network or participate within the network.

(7) Carbon Dating: The property that an existing data in a log over time accumulate work such that the work cannot be forged.

(8) Avoids Proof of Work (PoW): Mostly the use of PoW by some log is to derive consensus on contents of the blocks in a decentralized system, mint coins where applicable and rate limit the time for introduction of new information into the log. This property holds for a given log if the log can achieve any of the listed above without the use of PoW.

(9) Privacy Preserving: This property holds if the contents of the log are visible but encoded in a form that makes it impossible to derive any meaningful data from it to ensure that personal or identifiable information stored on the log are not visible to the users.

(10) Publicly Auditable: The log is public and available to be queried by users, auditors or monitors who want to audit the log to verify that the log is consistent.

(11) Decentralized: In this content, a decentralized system would be defined by the least number of people needed to break the system. The greater the number, the more decentralized the system is and the lesser the number needed the more distributed the system is. If only one party needs to be compromised, then the system is classified as a centralized system.

(12) Do not store entire log: A user might need to store the entire log to be able to audit or verify information stored in the log depending on the structure of the log. Some logs require users to store the entire log to make a full verification and some are required to store only the roots or headers.

(13) Resilient to attack: The more decentralized the system is the harder it is to corrupt the system as this would inevitably mean corrupting very single user in the system.

(14) Accountability: The ability to identity an entity responsible for a misbehavior and apply appropriate discipline.

See Appendix A for the details of the evaluation itself.

Chapter 3

Blockchains and voting

This chapter references and excerpts work (under review) co-authored with Yomna Nasser, Jeremy Clark and Peter Y A Ryan.

3.1 Introductory Remark

The blockchain a novel data structure and consensus mechanism designed for the decentralized currency Bitcoin, is currently being considered, exuberantly, for new applications in the financial and technology sectors. A steady stream of whitepapers from well-established banks (JPMorgan Chase, Deutsche, Barclays, Citi), professional service networks (PwC, Deloitte, EY, KPMG), and technology companies (IBM, Microsoft) explore how blockchains and distributed ledgers can create new digital assets, smart contracts, and provenance systems that enable direct interactions between participants, eschewing traditional intermediaries.

With all this attention (and some would say "hype") on new blockchain applications, it is unsurprising that a number of researchers, developers, journalists, and start-ups have posited blockchain technology as the missing link for transparent, verifiable election systems. Are they really? We approach this question in a systematic way from both directions: (i) beginning with a blockchain, what challenges arise from layering a voting system on top, and (ii) examining existing verifiable voting systems, how might a blockchain augment their properties? Our conclusion will not please those looking for one-armed researchers: blockchains are a useful augmentation to verifiable voting in some circumstances and may introduce interesting ways of voting in non-traditional settings; but on the other hand, blockchains are not a panacea. For current public sector elections in particular, any holistic approach to security will at most use a blockchain as a supporting component in a much larger system, if at all.

In this work, we do not propose a new system. Rather we systematized existing knowledge on blockchain technology and verifiable voting technology, and showcase what the landscape looks like for their composition. It is not a survey, per se, because we are commenting on future work, informed by past work but not summarizing it. Our goal is to provide a justified research agenda by bringing into focus where there is potential, and pruning out what seem to be dead-ends.

3.2 An abbreviated history of the blockchain

At the 1990 CRYPTO conference, a data-structure for appending a series of temporally-ordered messages to each other was introduced by Haber and Stornetta [22] and subsequently refined (alongside others[4]) . The later designs from this literature propose that a set of messages are aggregated into a single value using a cryptographic primitive (hash function) called a Merkle or Hash tree, such that one can efficiently prove a given message is fingerprinted by this value (called a Merkle root). If the root is widely witnessed (companies like Surety have published such roots in the Financial Times and the New York Times), the messages are indefinitely binded by the value. If a new set of messages is produced, can be locked in with the previous messages by hashing together its root with the previous roots to produce a new value called the header.

Two years later at CRYPTO, Dwork and Naor proposed a system for combating spam where the sender of an email would be required to compute the solution to a moderately hard puzzle (e.g., something that would take a normal computer several seconds or minutes) before sending the email [17]. However, verification of the correctness of the solution is relatively trivial and messages without a correct solution are ignored. Hashcash [2] was an instantiation of this idea where the puzzle involved computing the hash of the email (and metadata) iteratively, each time changing a meaningless counter included in the hash input, until the output of the hash function happens to have a specified number of leading zeros.

In late 2008, Satoshi Nakamoto (the pseudonym of a not-yet-identified developer) posted a whitepaper to a cryptography mailing list outlining a new digital cash system called Bitcoin [32]. By this time, many attempts at commercializing some form of cryptographic digital cash had been made, none reaching commercial success. The design of Bitcoin is quite different in several regards from previous work, but one vital design decision that distinguishes it from nearly all previous proposals is having a publicly available ledger that records every transaction. The ledger takes the form of the Haber-Stornetta data-structure: a set of transactions (called a block) are accumulated into a root, and the root is hashed with the previous block header to create a new block header (the chain of block headers being called the blockchain). Nakamoto's subtle but incredibly insightful twist is that each time a new block header is produced, it must integrate the solution to a moderately hard puzzle (the idea of Dwork and Naor).

3.2.1 A novel consensus mechanism

It is difficult to succinctly motivate and describe everything the addition of the puzzle enables. We refer the interested reader to a more thorough survey [25], but in short, Nakamoto's blockchain allows an open network of computers to update the blockchain in a way that (1) no node is in charge (no centralization), (2) all nodes agree on the contents of the ledger (a common consensus), and (3) nodes are incentivized to do the work of updating and maintaining the ledger (via mining).

A blockchain of length L includes the solutions to L puzzles, and these solutions will add up to a considerable amount of computational work. The puzzle difficulty adjusts periodically(approximately every 2016 blocks) to keep the average solution time at 10 minutes. If nodes see more than one valid version of the blockchain, they will first prioritize the chain with the most work (the longest chain), and then if chains with equal work co-exist, whichever they saw first. Once a node has prioritized a chain, it will assemble a block of valid transactions (checking for double-spends) and attempt to add it to the chain by solving the puzzle. All nodes are constantly doing this, and the first node to find a solution will broadcast it. This new chain should now be prioritized by all nodes (since it is one block longer than any other prioritized chains). The node that solves the block gets a reward and certain fees affiliated with the transactions they chose to put into the block. To ensure they capitalize on this,which in turn secures the blockchain, they must ensure their chain (both their block and each previous block) contains only valid transactions or the network will reject it. This incentivizes every node to validate every block of the chain they prioritize.

If a malicious node were to change or drop a past transaction in a past block, the root of this block would also change and, subsequently, every blockheader that follows it. Thus to get this modified chain prioritized, the attacker would have to "catch up" to the length of the current blockchain and could only do so if it had more computational power relative to rest of the network. There are important nuances to this process of forming a consensus, but generally it is the case that all nodes will agree on the contents of the blockchain from beginning to nearly the end, with some disagreement over the last few blocks to the tail of the chain [21]. For this reason, it is advised that one waits until a transaction is six blocks from the last block before considering it effectively immutable [32]. It is also generally the case that any valid transaction will eventually be included in the blockchain [21].

To build a currency on top of a blockchain, users chose a set of unrelated signing keys, for a digital signature scheme (currently ECDSA), and use the public keys as ”addresses” for receiving payments. No other identifying information is used. New currency enters the system via the nodes that solve the blocks (they specify an address they own in their solution) and currency can be sent in any amount to any other public key by digitally signing a transaction specifying the details (requiring the signature counters theft). The transaction is broadcast to the network and eventually added to the blockchain, which has a publicly verifiable history of every transaction (ensuring no double spending) and is generally immutable (preventing any future revocation of the transaction). This is a simplification of Bitcoin but a sufficient description for our purposes.

3.2.2 Distributed ledgers

It is useful to think of the blockchain as both a data-structure and a consensus mechanism that enables agreement on a ledger of messages amongst a set of mutually distrustful nodes, premised on the assumption that the adversary does not own a considerable fraction of the computational power on the network (e.g., half is sufficient to eventually rewrite the ledger and a quarter is sufficient to attack subtler properties [19]). The way consensus is formed, through a network of devices solving a computational puzzle as fast as they can, is considered by many to be wasteful and not desirable. Many of the companies exploring new blockchain applications are also considering how trusted entities can be added into the system to avoid computational puzzles, shorten block update times from 10 minutes (in Bitcoin) to seconds, and govern network join/write/read privileges. We use the term distributed ledger to refer to the family of technologies that includes both Bitcoin’s blockchain and these relaxed variants. Ironically, most distributed ledger technology strips away all of

Nakamoto's novel insights, and could have been deployed using 1990s' state-of-the-art knowledge.

3.2.3 Verifiable voting and digital cash are old siblings

Both digital currency systems and electronic voting systems have a long intertwined history in the cryptography literature. In fact, both originate with the same person: David Chaum. Chaum proposed the first cryptographic voting system in 1981, based on his mix network protocol [10]. The next year, he proposed the first cryptographic payment system based on his blind signature primitive [8]. Blind signatures were later studied extensively for voting (the hallmark paper being [20]), and cryptographic mixing has been used in digital cash, recently to anonymize Bitcoin transactions [6], [35]]. The shared history does not end there, with many cryptographic primitives and protocols being utilized in both payments and voting: zero knowledge proofs, ring signatures, digital credentials, homomorphic encryption, algebraic MACs, and on and on. We can now add blockchains to the list.

A cryptographic or end-to-end verifiable (E2E) voting system is one that provides a proof to each voter that her ballot was included, unmodified, in the final tally (even in the face of a corrupt election authority and malicious vote capture equipment). This proof generally consists of an obfuscated form of the voter's ballot choices, called a receipt, as well as public data about all recorded receipts and the tallying process that can be validated by anyone who wants (even those who did not vote). It is a strict condition of cryptographic voting systems that this proof leaks no information about how the voter voted, beyond what can be inferred from the tally alone, including the case where the voter deliberately crafts her ballot to leak how she voted. For in-person voting, we assume the voter's actions cannot be observed in the voting booth. For online or remote voting, no such assumption can be made and it becomes quite challenging (but theoretically possible [26]) to provide voter privacy when anyone can look over her shoulder and coerce/pay her to make certain choices.

3.3 A toy blockchain voting system

We now describe a simple voting system, layered on top of Bitcoin, to stack it up against the properties of a true E2E voting system. This is not a proposal for a system. Rather it is a pedagogic device we will tweak to showcase both the strengths and weaknesses of blockchains. It is not purely pedagogical however, as it is the core protocol of a number of blockchain voting projects including Swarm, Ballotchain, BitCongress, Blockchain Voting Machine.

Voters register a Bitcoin address with the election authority (EA). The EA publishes a list of addresses but does not list which address belongs to which voter. Each candidate also publishes an address. Voters then cast a ballot by sending a small payment to their selected candidate. Any deviation from the voting rules (e.g., one vote per voter) can be seen by inspecting the blockchain, and the tally is visible by inspecting the candidate's received payments.

This scheme has one major benefit and several drawbacks. The main benefit is that when voters cast their ballots, they are doing so to a decentralized global network of computers where at least some will have no affiliation to the election authority and will include the transaction in their blocks.

The first drawback of the scheme is that the registration authority knows the mapping between voter identities and keys, and the mapping between keys and candidates voted for is public, so the registration authority can break voter privacy. Known cryptographic techniques can address this in the following way. The voter list consists of real voter identities and each voter adds a public key encryption of her address to the list beside her name. A set of mutually distrustful parties takes each encrypted address, with the real identity left behind, and shuffles it into a new location in the list and then ”obfuscates” the value so that it cannot be recognized but still decrypts to the same value. Once the list is shuffled by each party, the complete set is decrypted with a decryption key shared amongst them. This system is essentially what Chaum originally proposed in 1981! In his scheme, voters chose random signing keys that are shuffled and revealed in the first phase, and used to sign ballots in the second phase [10]. A modern variation could use ECDSA signatures and provably correct shuffling [23].

A fairly equivalent approach could have voters approach the registration authority in order to obtain a signature on their address. By using a “blind signature” scheme, the voter can obfuscate (blind) the address such that the authority just sees random-looking data when it signs, and then de-obfuscate (unblind) the message and signature in tandem such that the signature is properly formed relative to the original address. This is the basis of many voting systems (such as Follow My Vote) and [20], and is largely equivalent to the voter list approach above with the drawback that once a signature is issued, it becomes invisible and cannot be revoked (if a voter becomes ineligible).Also no one can enumerate how many signatures an EA has handed out. A third is that the EA can decide not to respond to a voter who wants to register his public key after ”Unblinding” so that he can populate the list with public addresses that he controls.

3.3.1 Tokens and coins

Because bitcoin payments can be traced on the blockchain, it is possible to flag a small amount of bitcoin to serve as a digital token for some other kind of asset. These tokens can be transacted between parties in the same way as Bitcoin itself (and possibly in new ways). Colored coins, Counterparty and MasterCoin rely on this idea. While a ballot can be tokenized, the difference is largely semantic. In the toy scheme, anyone can send bitcoins to the candidate addresses and the system will have to use a list of authorized voters to filter out spam transactions from the real ones. Tokens do not solve this issue as the process for checking a token's validity is also consulting a list of coins that have been tokenized. In a closed distributed ledger, submitting a transaction could require authorization. This would eliminate spam but would reintroduce trust in the network for censorship-resistance and consensus.

3.3.2 A running tally

Even with one of the address anonymization add-ons, our toy blockchain voting scheme has the property that votes can be seen as they come in, enabling a visible running tally for each candidate. From a transparency perspective, this can be either a feature or a bug. Some blockchain voting projects (Follow My Vote) advocate this as a beneficial feature. They argue that when combined with the ability to change one's vote (this could be accomplished in the system above by giving the voter many tokens with the policy that the ”last vote” counts), a new type of scoring protocol emerges where voters can cast a ballot for their most favored candidate and then manually intervene if it seems they are too far from winning midway through the voting period. In response, we would argue that ranked choice voting is a more direct way of achieving this, and one that removes both the voter's guesswork and manual intervention. Further, open tallies are a ”bug” in one major sense: they are illegal in essentially all governmental elections today. A second issue concerns ballot secrecy: we know from paper audit trails (added to electronic voting machines) that writing votes into a ledger in the same order as they are cast can reveal how voters voted through simple timing analysis.

3.3.3 Hiding voter choices

These two drawbacks, legality and ballot secrecy, motivates us to consider how the tally might be kept confidential until voting closes. Once again, the E2E literature offers numerous solutions to this problem and we highlight two general approaches in applying the literature to this issue.

The first approach to hiding the vote itself is to have voters submit their votes as a message instead of moving tokens to a candidate's address. Bitcoin offers a transaction called OP_RETURN that allows a user to burn a small amount of Bitcoin to embed 80 bytes of data into the blockchain. The data could be a public key encryption of the vote with the EA's public key, to be tallied after the election in a privacy preserving manner (e.g., using additively homomorphic encryption [14] or with a verifiable mix-network [37]). Alternatively, the set of voters might engage in a pre-election protocol that produces for each voter a secret random number they can add to their vote to mask it, such that the sum of all mask values is zero [28]. When the last vote is submitted, the blinding factors all cancel out leaving just the sum of the votes, eliminating the need for an EA. In both cases, this is the core idea of a much larger protocol that needs additional steps and proofs to achieve all the desired properties of an E2E system. A system of the second type has been recently proposed and implemented [30].

The key question this raises is if such a system is really a "blockchain voting" system any more given it is using the blockchain in a very limited fashion: for posting publicly viewable messages into a ledger. Toward an answer, consider a second approach that can leverage voting through financial transactions, as in our blockchain voting example, while preserving the privacy of the vote until after the election is complete. The main obstacle to accomplishing this is the known, one-to-one mapping between candidates and addresses. Instead, each ballot could have a unique set of one-time use addresses for each candidate. This is closely related to systems like Scantegrity [9] where each candidate on each ballot has a unique code associated with them: in our case, the code happens to be a Bitcoin address. A largely equivalent approach would be to distributed ballots that have the order of candidates shuffled, and have a set of addresses corresponding to the first candidate on the list, the second, etc. Such a system would correspond to Pret a Voter or vVote [36]. The backend of any of these systems would be able to verifiably map votes back to the correct candidates while preserving ballot secrecy.

Our key observation can now be made. There does not seem to be any difference between a voter sending a token to an address affiliated with their selection with a standard Bitcoin transaction, and simply writing the selection in an OP_RETURN message. The latter can do everything the former can (and more). Thus the ability to conduct transactions does not seem necessary for securing a voting system. In attempting to build a blockchain voting system, we have to layer so much additional cryptography on top that it effectively becomes a normal end-to-end voting system from the literature with the one modification that public messages are recorded on a blockchain. This begets an important follow-up question: where do E2E voting systems propose storing their public messages and is it any better than a blockchain?

3.3.4 The mythical bulletin board

Virtually every E2E voting system assumes the existence of a ”bulletin board” which was first formalized by Benaloh and Tuinstra [5] and is considered to be an append-only, broadcast channel. The vast majority of papers adopt this as an assumption without specifying exactly how a bulletin board should be constructed. When E2E elections have been run in the real world (e.g., Scantegrity, vVote, Helios, etc), the bulletin board is typically implemented as a website with data signed by the election authority, perhaps with some replication by other interested parties.

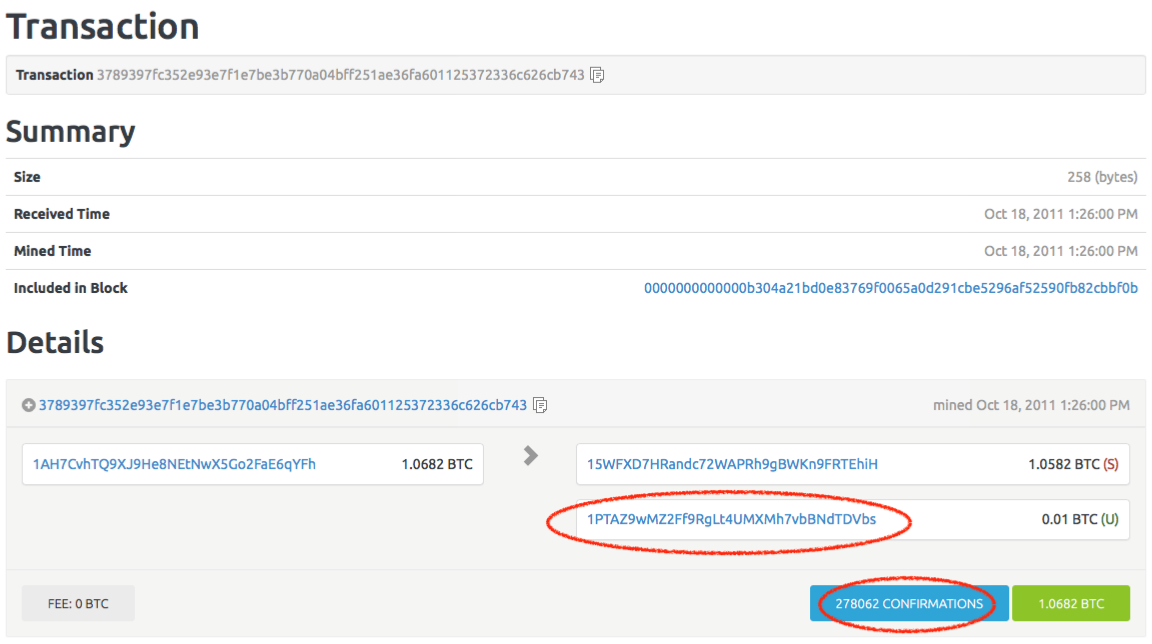
Such an implementation is neither strictly append-only nor a broadcast: the data can be modified or reordered at any time the EA choses, the EA can equivocate on what is stored on the bulletin board by sending different records to different voters, and the EA can play gatekeeper and drop messages it does not want published. Of course, the EA might get caught modifying the bulletin board, and could be held accountable because it signs every version, however this is a detection mechanism more than a prevention mechanism.

By contrast, embedding messages into a popular blockchain offers several improvements. First, there is no single entity to serve as gatekeeper and no successful cases of censorship have been observed in Bitcoin's history. Once a record is incorporated several blocks from the tail-end of the blockchain, it cannot be feasibly modified, and everyone holding the longest chain will see exactly the same records in this part of the chain (this is called a common prefix in the blockchain literature [21]). For these reasons, it is our opinion that a blockchain is the best bulletin board implementation we are aware of.

3.3.5 Carbon dating messages

Blockchains have one more trick that a traditional bulletin board cannot do. Because updating the blockchain requires computational work, a message that is X blocks from the tail-end of the blockchain was embedded X puzzles ago in time. Computing the solution to X puzzles takes a measurable amount of time. We might not be able to be very precise, since the computational power of the network varies, but we cannot shortcut months of the entire network's computational work in a few days. Thus a message embedded in a blockchain accumulates work over time, much like a fossil accumulates carbon, and this work cannot be faked even if the entire network is malicious.

This unusual property was observed by Clark and Essex [12], two contributors to the Scantegrity system. In Scantegrity, certain commitments are made prior to the election and if a corrupt election authority changed and backdated them after the election, without being noticed, the soundness of the tally is no longer guaranteed. Naturally voters may only become interested in verification after some unexpected election result, and such a voter has no way to know from a bulletin board whether the pre-election commitments were really



Transaction

Transaction 3789397fc352e93e7f1e7be3b770a04bff251ae36fa601125372336c626cb743

Summary

Size	258 (bytes)
Received Time	Oct 18, 2011 1:26:00 PM
Mined Time	Oct 18, 2011 1:26:00 PM
Included in Block	0000000000000b304a21bd0e83769f0065a0d291cbe5296af52590fb82cbbf0b

Details

3789397fc352e93e7f1e7be3b770a04bff251ae36fa601125372336c626cb743 mined Oct 18, 2011 1:26:00 PM

1AH7CvhTQ9XJ9He8NEtNwX5Go2FaE6qYFh	1.0682 BTC	15WFXD7HRandc72WAPRh9gBWKn9FRTEhiH	1.0582 BTC (S)
		1PTAZ9wMZ2Ff9RgLt4UMXMh7vbBNdTDVbs	0.01 BTC (U)

FEE: 0 BTC

278062 CONFIRMATIONS

1.0682 BTC

Figure 3.1: A screenshot from Blockexplorer showing the Scantegrity pre-election commitments embedded in Bitcoin's blockchain prior to the 2011 municipal election in Takoma Park, MD. At the time of writing, this message has been extended by 278062 blocks (each block requiring an average of 10 minutes of computation work from the Bitcoin network). The commitment value itself was embedded as a Bitcoin address since OP_RETURN was not supported at the time. The 0.01 BTC at this address is thus unspendable.

made before the election or not. Thus, the researchers embedded the pre-election commitments for the 2011 municipal election in Takoma Park, MD (which was using Scantegrity) into Bitcoin's blockchain so that voters after the election could be reasonably sure that the pre-election commitments must have really been made before the election, since by the time of the election, they had acquired a month of computation work.

3.4 What would deployment look like?

If a blockchain were deployed as a bulletin board in an election, how would that look? For in-person voting systems, voters would use standard voting technology. E2E systems have been built on both optical scan and direct-recording electronic (DRE) architectures.

In this case, the EA equipment (whether the devices the voter operates or some central tabulator) would post the events to the blockchain. The voter would retain some sort of privacy-preserving receipt (e.g., a confirmation code, ciphertext, marked ballot position, etc.) that they would use to cross-check with the data posted on the blockchain. Thus, their first and only interaction with the blockchain would come after the election for the purposes of an audit. This would require a computer program or website they trust. While the blockchain itself will not equivocate on what is stored, the voter's computational device can certainly display the wrong information (the voter cannot compute hashes in her head to detect something is amiss). The benefit of using Bitcoin's blockchain is the number of third party software and web-based tools for reading data from the blockchain.

In an online voting setting, voters cast their votes from their device which requires write access to the blockchain in addition to read access. The voter's device might not be able to obtain the correct software (or client-side scripts) due to a network attack and well-crafted malware can always interfere with a voter's ability to vote even if the network connections are secured with HTTPS. A malicious device can change a voter's selections, while simulating the process of casting the correct choices, and then simulate the receipt check should the voter perform one after the election from the same device.

The voting literature has a number of schemes, starting with Chaum's SureVote, where voters are mailed random-looking codes corresponding to each candidate to type into their devices so the device has no way of knowing how it might modify the vote. These solutions are largely tangential to the use of a blockchain but we make one remark. In an advanced code voting system like Remotegrity [43], the voter and EA exchange codes until the voter is satisfied and then sends a final lock-in code. A blockchain can provide assurances that this lock-in will be eventually incorporated if it is successfully broadcast.

3.4.1 Can a human use that?

End-to-end voting systems have evolved over time to systems where as much cryptographic detail is swept under the carpet as possible, especially during voter interactions. In any voting system that requires the voters themselves to authorize a blockchain transaction, the voter will need to safely and reliably store a cryptographic key pair. Key management is known to be difficult for voters whether for secure email or for Bitcoin itself [18]. It is generally not sufficient to use shorter passwords (cryptographically expanding them into keys) as the public keys will be placed on a public blockchain where they can be subject to offline brute-force attacks. In fact, many Bitcoin thefts have occurred because of short password-derived keys. Any system that authenticates voters using cryptographic keys needs to carefully consider usability factors.

3.4.2 DRE audit logs

In E2E voting systems, it is sufficient to only write critical data like voter receipts to the bulletin board. However if an incident were to trigger an E2E system to not validate the final result, election officials would appreciate a finer-grained log of events to uncover issues. In fact, non-verifiable voting technology like DRE (direct-recording electronic) machines record a detailed electronic log of every operation they do across an election. DRE logs are critical in post-election audits. An issue is that the potentially faulty or malicious DRE generates and maintains its own log itself. In 2007 and in subsequent work, Sandler and Wallach propose that DREs broadcast logged events to a private network of nodes (called VoteBoxes) for recordings in a Haber-Stornetta type of data-structure [38]. It would be quite natural to implement their Auditorium system with a private blockchain run between VoteBoxes.

3.4.3 Voting through smart contracts

Blockchain projects like Ethereum enable verbose smart contracts to be expressed and enforced on a blockchain. This opens eligibility to anyone matching a specified algorithmic description. It is hard to imagine all the possible forms this could take but it is one key area for future research to explore. So far in this section, we have concentrated on the use-case common to all governmental elections but other elections are imaginable.

Assume ownership shares in a company are issued on a blockchain (as was recently approved by the United States Security Exchange Commission for Overstock) and a shareholder vote occurs, shareholders could cast ballots that are weighted by how many shares they own (their stake) and they could even do so without revealing which shares belong to which shareholder. The outcome of a shareholder election might trigger certain outcomes on the same blockchain: financial transactions, stocks to split, new shares, or a dividend to be paid out. NASDAQ is experimenting with this type of system in Estonia.

A voting system could also be designed to give one vote per unit of computational work performed. Some Bitcoin users generate for themselves ”vanity addresses” where their otherwise random Bitcoin address contains a certain prefix of characters. They do this by generating new addresses as fast as they can until one happens to have the right prefix. An election could be held where votes are only counted from addresses containing a certain prefix (the longer the prefix, the exponentially more work it requires to generate). This could also be layered onto a ”one vote per voter” scheme as a spam deterrent. The idea of eligibility based on computation effort is actually used indirectly in Bitcoin today. The Bitcoin community sometimes holds polls on protocol changes and the network nodes that solve the blocks include their vote in their solved blocks, which skews the result toward the side doing the most work solving blocks.

Some countries, like Canada, have a per-vote subsidy where a set amount of money is paid to political parties for each vote they receive, this could be automatically redeemable after an election. A country with mandatory voting might collect a deposit fee that is later refunded upon receiving a vote. More generally, voters might post some money as a surety that is returned only if they behave correctly within the protocol [30]. This blend of voting and payments could also be used for nefarious purposes, like automating vote buying (c.f., [27]). Bets on an election outcome could be automatically settled. While the outcome of an election can always be reported to a blockchain by a trusted party (called an ”oracle” in the Bitcoin community), having an on-blockchain election can enable direct observation of the outcome by other scripts running on the blockchain.

To date, both Bitcoin and Ethereum have inherent scalability issues. Bitcoin only supports a maximum of 7 transactions per second [32] and each transaction dedicates 80 bytes for storing arbitrary data. On the other hand, Ethereum explicitly measures computation and storage using a gas metric, and the network limits the gas that can be consumed by its users as earlier mentioned. Therefore, these Blockchains cannot readily support storing the data or enforcing the voting protocols execution for national scale elections[30].

A smart based voting system offer vital benefits over traditional voting system in terms of Transparency, public verifiablity, low cost and more. The important critical can be viewed from two angles. The Electoral body needs to ensure that votes originates from eligible voters and the voters in turn needs assurance that his vote is secret(ballot secrecy) and the votes are tallied without any manipulation. This two aspects are hard to achieve without the inclusion of a trusted party. However, there are current systems ([30]) that use smart based contract for other forms of elections outside national state elections.

3.5 Concluding Remarks

Satoshi Nakamoto’s blockchain is undeniably a breakthrough for decentralized digital cash, and it may prove useful in disintermediating other protocols and procedures. However when it comes to voting, it seems the most useful contribution a blockchain can provide is a broadcast channel for cryptographic voting systems. In this role, blockchains are not only sufficient but actually provide some advanced properties, like carbon dating, that cannot be directly achieved with a standard bulletin board protocol. Blockchain-based voting systems also have the potential to entangle the eligibility for an election and the outcome of the election with other blockchain payments, assets and smart contracts in new and creative ways. While this is a somewhat abstract thought to end on, we are confident that researchers will recognize useful scenarios for this incredible flexibility.

Chapter 4

Lending Infrastructure

The role of lending in society is paramount for its economic stability and growth. Bitcoin was developed, in part it seems, to replace central banks. What does a central bank actually do? First and foremost, it is a bank and so it accepts deposits and pays interest. It can pay interest because deposited money can be lent with interest (the same principal applies to retail banks). Thus interest rates are a product of lending.

Currently, most central banks actively influence the interest rates of the country (specifically the interbank loan rate), after decades of experimentation with managing different economic indicators, because it has proven the most reliable for governing the economy. It controls the interest rate by purchasing or selling bonds — loans to the government packaged up as financial instruments. This trading is generally with its member banks and it, respectively, increases or constricts these banks' access to cash. Banks loan when cash is in excess (borrow when cash is restricted) and this supply-side (demand-side) pressure on cash pushes interest rates down (up). Thus the central bank is targeting a lending-derived metric by purchasing loan-based instruments.

Finally, when Alice deposits money (say \$1000) in a retail bank and the retail bank lends it to Bob (say \$900), Alice will claim she has \$1000 and Bob will show you his \$900 and so the amount of money in the economy somehow increased to \$1900 via lending. This money multiplier effect is a direct product of lending. (Of course, if Alice and Bob both demanded their money at the same time, the bank would have a liquidity problem and this is a further function of the central bank: it could serve as a lender of last resort).

Given the permeance of lending in our economy, what can we say about lending in a cryptocurrency market? Bitcoin has virtually no lending infrastructure. This is likely due to its consistent appreciation in value over time. Taking a loan out in Bitcoin requires a belief that it's value will remain stable or decrease over time, which is not supported by historical trends. While investors might be thrilled with Bitcoin's appreciation, it inhibits Bitcoin from operating as cash (as a stable store of value). It prevents lending and it promotes hoarding and until these issues are dealt with, Bitcoin will never really succeed at its original goal of being digital money.

In this chapter, we introduce a lending infrastructure for cryptocurrencies as a stepping stone towards addressing the shorting comings of cryptocurrencies in general. The main issue in lending is counterparty risk (Alice lends to Bob who does not pay Alice back). If a loan is really a loan — the money is actually transferred to Bob's full control — this seems like an unsolvable problem and certainly not a problem one can 'blockchain' their way out of. That said, blockchain can help in a few ways. First it can introduce transparency which can establish reputations and can make at clear when a loan default (credit event) takes place. Since blockchains can execute smart contracts, a credit event can automatically and atomically trigger pre-arranged actions when credit events occur: insurance payments, collateral expenditures, selling the loan for cash (a credit default swap), and others. In this chapter, we build a general platform, as well as implement several instances of credit event reactions used in finance today, but by virtue of the blockchain, implement them without a trusted intermediary.

4.1 Introduction to smart contracts and decentralized apps

Contractual financial instruments such as loans, bonds, collateralized loans, and credit swaps can be automated in such a way as to make a breach of contract expensive for the defaulter and transparent for all. Szabo in his paper [40], gave an example, which we might consider to be the primitive ancestor of smart contracts—a humble vending machine. The machine accepts coins, and through a simple mechanism, dispense product per the displayed price. The vending machine is a contract with a bearer token: anybody with coins can participate in an exchange with the vending machine. The lockbox and other security mechanisms protect the stored coins and contents from attackers, sufficiently to allow profitable deployment of vending machines in a wide variety of areas [40].

Each transaction in Bitcoin references other unspent transactions output as inputs and creates outputs, that are inserted into the Bitcoin blockchain. These outputs can then be "spent" by other transactions. The creation of transactions is facilitated by a scripting language that is used for specifying conditions needed be met to spend transaction outputs [1]. Since the birth of Bitcoin, other types of blockchain based systems have extended the scripting language beyond the realm of a cash systems, to enable an unimaginable variety of decentralized applications (or dapps) expressed on the blockchain.

Currently one of the hottest areas within cryptocurrencies, smart contracts, are agreements between two or more parties that can be automatically enforced without the need for an intermediary. Today's leading smart-contract platform is Ethereum, whose blockchain stores long-lived programs, called contracts, and their associated state, which includes both data and currency (ether). These programs are immutable (except through a violation of consensus), just as data on the blockchain is, and users interact with them with the guarantee that the program will execute exactly as specified [3].

In a formal approach, smart contract refers precisely to the use of a step-by-step set of machine readable operations to formulate, validate and execute an agreement between parties. Where a standard contract is drawn up using a legalese form of natural language, the conditions in smart contracts are expressed in code, comparable to (in fact, modelled after) programming languages like Java and Python.

4.1.1 Structure of a contract

Contracts in Ethereum are most often written in a high-level (developer friendly) language such as Serpent (Python-esque), Solidity (Java-esque) or LLL (Lisp-esque). This code is compiled into Ethereum's custom byte code that is interpreted by the Ethereum Virtual Machine (EVM). For a variety of reasons, Solidity has become the most widely used language for programming contracts in Etheruem. In this section, we illustrate a few basic concepts for Solidity.

In Solidity, there are four types of visibility for members (variables and functions): Private, Public, External, Internal. Private function or state variables allow the member to be accessed only by functions within the contract. Public functions can be accessed by any function either internal or external. An external function can be accessed from other functions but cannot be called internally. Internal functions or state variables can be accessed from the same contract or contracts that extend (via inheritance) the contract. Other object oriented concepts like abstract classes, interfaces, and function overriding are possible in Solidity. It is important to remember that everything that is inside a contract is place on an open blockchain, visible to all external observers. Making something private only prevents other contracts from accessing and modifying the information (allowing encapsulation of the code) but it will still be visible to all.

Every contract in solidity begins with a compiler version (pragma solidity $\hat{0}$.4.4;). This complies the code into assembly language that can be understood and interpreted by the Ethereum language.

State Variables: These are variables that are assigned a value that are stored in the contract storage permanently. For state variables, the default is internal and an eternal mode cannot be specified.

Algorithm 1 State Variable

```
pragma solidity ^0.4.4;
contract Loan {
    address Lender;
    address Debtor;
    uint256 Principal; }
```

Functions: Functions are units of codes that are executable. Functions can either be external, public, internal or private. They are by default public.

Algorithm 2 Function

```
pragma solidity ^0.4.4;
contract Loan {
function cancel(){
    if (msg.sender == Lender){
    Timestamp = block.number;
    suicide (Lender);
    }
}
}
```

Constructor: Every contract that has a function within the contract with the same name as the contract is termed a constructor, and it is called at the creation of the contract. This function is widely used to set and initialize basic parameter that would be needed within the contract's life span.

Function Modifiers: In Solidity, modifiers are used to restrict a function to be executed only when certain conditions are met. It is an efficient way of setting conditions across various function in order to avoid repetition. An example would be to set a function to be executed only by the owner of a function as seen in Algorithm 4.

Algorithm 3 Function

```
pragma solidity ^0.4.4;
contract Loan {
function Loan(){
        owner = msg.sender){
        Timestamp=block.number;
        }
}
}
```

Algorithm 4 Function Modifiers

```
pragma solidity ^0.4.4;
contract Loan {
        modifier onlyOwner()
        {
          if (msg.sender != owner) throw;
          _;
        }
        function serveBond (address investor) onlyOwner
        {
        if (status!= State.Initialized) throw;
        }
}
```

Events: A piece of code declared globally that can be placed inside a function that records on the blockchain any variable placed within its argument. This keeps a historical trace of a transaction from an address either controlled by a user or a contract. See Algorithm 5.

Algorithm 5 Event

```
pragma solidity ^0.4.4;
contract Loan {
        event Transfer(address indexed from, address indexed to, uint256 value);
}
```

Enum Types: In Solidity, multi-stage contracts can be formalized with a state machine. The state of a contract can be defined using Enum types. Because different functions can be executed in any order (including orders not anticipated by the developer) which could break the contract, states can be defined to allow an explicit state transition flow for the contract. See Algorithm 6.

Algorithm 6 Enum Type

```
pragma solidity ^0.4.4;
contract Loan {
    enumState Initialized, Release
    State status;
    function serveBond (address _investor)
    {
    if (status!= State.Release) throw;
    }
}
```

Structs Types: Struct is a way of grouping variables of different data types into a single container which can be treated like a single variable.

Algorithm 7 Struct

```
pragma solidity ^0.4.4;
contract Loan {
    struct Default
    {
     address Debtor;
     bool defaulted;
     uint256 timestamp;
     }
}
```

4.2 A Simple Loan as Smart Contract

A simple loan can be illustrated as: Alice lends Bob a certain amount (principal) of money which Bob uses in a project of his choosing that generates a return (profit) such that Bob gives Alice an interest over the term of the loan and then the principal at the end of the term. There is always a trade-off as regards to loans in general, the riskier the project Bob intends to use the loan for the higher the interest Alice would receive and the greater the risk Bob might not pay back the principal. There are key factors every lender puts forward to the borrower—how much, how long, how risky, will the borrower pay back at all, how to handle early payments, etc. All these factors are usually handled by the banks. However, due to the level of bureaucracy and regulations, lending without a central bank might be cost-effective.

After the financial crisis in 2008, US-based investors began to seek a better platform to invest their money which gave rise to peer to peer lending. This platform matches lenders directly with borrowers, charging borrower less interest fees and giving investors greater returns. A typical approach is a centralized webservice, backed by brokers, that match a lender to a pool of borrowers. The lender chooses any number of borrowers according to their preference. In the case, the service would enable the lender to make separate loans for the chosen borrowers. However, needless to say, this approach comes with various logistical and risky drawbacks. First, the created loans might have different interest rates. Second, they might have different repayment dates. Third, the borrowers involved will have different risk profiles. Fourth, everything has to be managed by the lender who might be inexperienced in this area. These disadvantages are what a second approach tries to combat: here the lender lends money to a pool of borrowers where he gets a standardized interest rate. In this case all the aforementioned disadvantages of the first approach are handled by brokers or intermediaries. However, this approach does not eliminate all the risk involved as the platform might not invest the money in time, the lender remains liable if the peer to peer lending platform crashes, and so on. As peer to peer lending platforms became more popular, traditional institutions have invested heavily and carved out an intermediary role for themselves.

The success of the decentralized cryptocurrency Bitcoin stirred us towards exploring the potential of a decentralized lending platform on the blockchain with the hope that this new cryptocurrency would provide a better platform for investment. Although peer to peer lending on the blockchain does not eliminate the counter-party risk associated with standard peer to peer lending, it brings transparency to the system both from the lender's and borrower's view. In this system, all transactions are recorded in chronological order and in near real time. In addition, the user stays in control of his money and the money can be transferred directly to the borrower without it being held for any period of time by any intermediary party, reducing the operational risk of the transaction.

The rest of this chapter discusses a walk through towards the deployment of the lending infrastructure.

4.3 A Simple Bond

A loan is a general term use to describe any form of debt. A loan can be in any of these form: peer to peer lending as discussed in the previous section, banking where banks take loans from a central bank or are involved in a repurchase agreement (RePo), or a company issued debt instruments, commonly referred to as commercial paper or bonds. For this thesis, we focus on peer to peer lending and bonds.

Commonly, shares of a company represent ownership of the company which implies that an individual owns a certain percentage of the capital or equity. Let's say that company A has lots of shares sold to investors and the company still wants to raise money for expansion. One option is to issue new shares but this would dilute the existing shares. Another option is to take a loan from a bank but due to the bureaucracy associated with banks, the company might not get a good interest rate. A third option is that a company can issue bonds to individual investors. A bond is a promise to pay certain amounts of money in the future according to some fixed schedule and it is sold for a present amount of money (typically less than the amount it will eventually repay: the difference being thought of as interest). These individuals could be anybody in the sense that the company do not need to know who would eventually buy the bond. The first two options are centralized but the last option is more decentralized in the sense that the people who hold the bonds do not need to have a direct relationship with the company. Hence incorporating this into a blockchain platform seems promising.

In issuing bonds, organization faces one basic problem: there is no guarantee that anyone will eventually buy the bonds. A decision has to be made as to what amount raised from selling the bonds would be acceptable. If the money raised is not enough for the project at hand, then the money could either be returned to the investors or augmented from another income source. For a smart bond, this would mean that the investors should be able to withdraw their money from the smart contract if after the deadline, the cap for the amount to be raised is not reached and no other source of fund is used. This operates the same way crowdfunding works.

Bonds can either be registered or not registered. One option is to have a registered bond in the name of the bond issuer and in this case, from a development standpoint, there is no paper (object) that has any terms of the bond, just the issuer's public key stored in the smart contract. The other option is that a token is issued to investors of the bond such that the token can be traded with anyone. At the maturity date, the interest is paid to whoever holds the bonds. This is generally referred to as bearer bonds.

A bond is simply a debt obligation where investors who buy corporate bonds are lending money to the company that issued the bond. The company in return makes a legal commitment to pay interest to the investors on the principal and return the principal at maturity date. Generally, like every other debt obligation, bonds differ according to the type of interest payment they offer. It is common for many types of bonds to pay a fixed interest rate throughout the duration of the bond. This rate is commonly referred to as a coupon rate (with coupon payments given at fixed intervals). Since the rate stays fixed regardless of any change in the market interest rate or in the value of the underlying currency, trading in these bonds is implicitly taking a financial position on future interest and inflation rates.

Floating payments on the other hand are another type of interest payment where floating rates are reset at given times. Intuitively, these bonds adjust their interest rate to changes in the market interest rates. Another type of bond gives no interest rate until the term is due. These are generally referred to as zero-coupon rates as they do not make any periodic payment. As a result, at maturity, the bonds return a single payment that is higher (in nominal terms) than the purchasing price. In the course of this thesis, the coupon payment and the zero-coupon payment are explored.

4.4 Protection

Lending in a decentralized system using any cryptocurrency poses two basic challenges: counter party risk and changes in the value of the underlying currency.

4.4.1 Challenge 1: counter party risk

On a light note, there can exist a loan with no counterparty risk. If Alice 'loans' some amount of money to Charles but the money never leaves the smart contract, meaning Charles never claimed the money, there is no counterparty risk. This is a trivial construction of a loan but illustrates the notion that unless the money is received by a borrower, there is no counterparty risk. Once the borrower withdraws the money, there is the counterparty risk and therefore, every meaningful loan has counterparty risk. In the event that Charles was not able to repay the loan at the maturity date, this is referred to as a Credit Event.

We reiterate that blockchains cannot prevent credit events, or broadly solve counterparty risk. Every financial instrument has counterparty risk. If you get an insurance, there is a risk the insurance would not be able to pay back or if one does a derivative, there is a risk that the other party never shows up to buy it at the price that they agreed on or if one goes for a mutual fund there is the risk that the mutual funds leaves with the money. The interesting fact about the blockchain is that it gives visibility and transparency in the occurrence of a credit event. This feature is what this thesis leverages.

In this light, in other to minimize the risk, a few actions can be taken. One would be for the borrower to post a collateral where the collateral is returned if no credit event occurs. This collateral can either be full or partial. In the case of a full collateral, it is usually an instrument different from cash (else the need for the loan is not apparent) which could be fungible or not, and liquid or not. Non-fungible instruments are uniquely valued. For example, a bill is said to be fungible as a currently circulated $20 bill would have the same value as any other $20 bill. However, a piece of jewelry given as a gift might be unique or have some circumstantial value (it was owned by a famous person) that an equivalent piece of jewelry might not have. Instruments are said to be of low-liquidity if they cannot be easily exchanged for money. For example, a house owner could put up his house as a collateral against a mortgage loan, which in many cases would be worth more than the loan, but turning this collateral into cash has a lot of market friction and would be hard to extract the full value of the house (hence why foreclosed houses often sell below market rates). By contrast, a collateral in the form of government bonds or company stocks might be readily turned into cash and are therefore highly liquid. Collateral that is liquid and fungible will give the lender the most options for arranging a loan, while collateral that is illiquid and non-fungible (e.g., a house) is the most limited (but still serviceable).

If Alice lends to Bob and Bob posts collateral, the collateral is not owned by either Alice or Bob for the duration of the loan. It stays locked up until the loan is resolved. Alice might get more utility from lending to Bob if she were able to use the collateral (sell it, lend it, etc.) while Bob held her cash. This assumes the collateral is fungible and works best when it is liquid. Central banks uses these arrangements when lending cash to member banks. It is known as a Repurchase Agreement (Repo). At the end of the term or maturity of the loan, the cash loaner must reacquire the collateral (if necessary) and return it to the borrower for the principal.

Collateral may be partial instead of full. This is generally not used directly in loans but might be used on, say, an entity providing insurance on a loan. If an insurance company insures a diverse set of about hundred loans, the chances of all of the them defaulting simultaneously might be small so the company will collateralize enough cash to cover (say) 4% default rate. This requires less cash than making all of the loan which is why the company would not simply issue the loans themselves.

The use of reputation is another action that could be taken to reduce risk. In this case, everyone could see that a default occurred and then no one would want to loan money to the defaulter. However, reputation does not work well on bitcoin because the nodes have pseudonymity and a user can have several identities. There is currently an ongoing research on integrating reputation with ethereum using Reptokens. Note that reputation requires visibility, which means your loan is on the blockchain for everyone to see. To buttress this point, the idea isn't that this would replace all loans and not everyone would be comfortable with this idea but this idea might be acceptable to an individual who wants to get some benefits in return (lower fees, a loan for subprime, unbanked, distrust banks, control of his/her money, faster transactions and so on). So therefore, the visibility offered by the blockchain system is seen either Benefical(valuable) or appalling(dreadful) depending on the perception of the individual.

A final action, hinted above, could be an insurance on the loan where a third party pays out the debt. In such case, everyone is aware of the identities of each party involved. A credit default swap is a type of insurance where anybody can issue/hold insurance on any debt with visible credit events. If there is a credit event, the lender (if holding the insurance) is made whole again by the insurer, and the insurer takes over the loan. When there are multiple issuers/holders of insurance, a more complex resolution is taken which is discussed later in this thesis.

Figure 4.1 gives a pictorial view of the different protection discussed above. One of the intent of this thesis is to explore the feasibility of the aforementioned ways of reducing risk on loans implemented on the blockchain. The challenges encountered during the implementation are discussed in later sections of the chapter.

4.4.2 Challenge 2: inflation/deflation

Bitcion and Ether have generally increased in value over time, a term referred to as deflation. This could be due to the fixed supply although no one really knows why the price moves. With fiat currencies, the supply of money tends to grow over time: the central bank uses newly created money to purchase bonds to influence market rates. The current policy is for the value of money to remain relatively constant in the short-term and inflate (decrease in value) slowly over a longer term. Mild inflation discourages hoarding cash and allows companies to lower employee wages without actually lowering the nominal amount of their salary.

Realistically, for a currency, one would not want the value to go up at a rapid rate as seen presently with Bitcoin and Ethereum. This is because two problems arise: first, a bitcoin owner would not want to spend the money as the value would increase faster than what was purchased with it and the owner would be at a lost. A popular example is the man who bought a pizza for 5000BTC which was worth about $5 back then but is now over millions dollars. Another case is that one would never want to take a loan in bitcoin. Imagine one took a past loan of 35BTC, and exchanged it into CAD to spend it. At the maturity of the loan, bitcoin would have increased in value such that the repayment of the 35BTC might result in a repayment of an amount ten or a hundred times larger in terms of CAD.

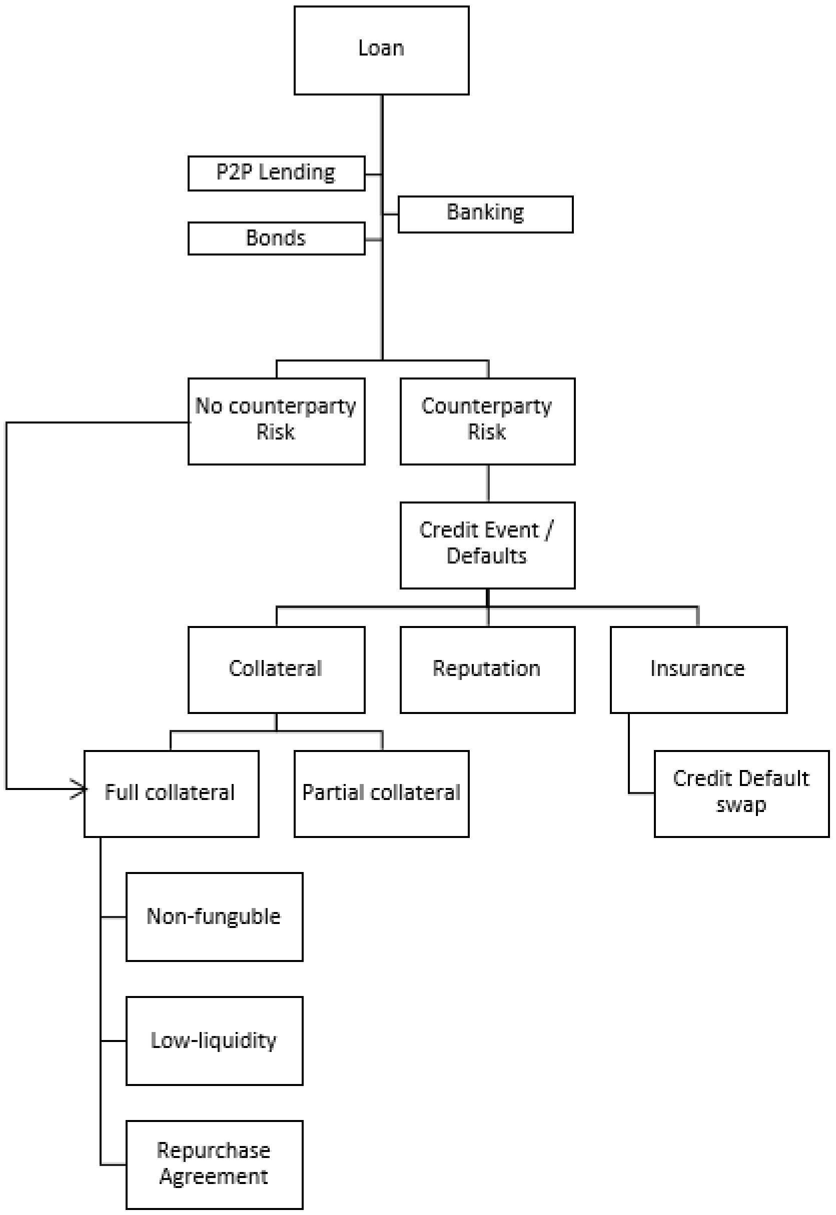


Figure 4.1: A diagrammatic representation of a loan

There is a notion known as inflation adjusted bonds where the purchasing price of the bond is what is reflected, irrespective of the inflation. We could approach lending in cryptocurrencies similarly (or simply make unit of account of the loan be that of a fiat currency like CAD). However how would the nodes in blockchain network come to know the exchange rate of a currency like Ether?

Consider that there is a company that issues a loan (say a bond) for some debt denominated in CAD. The counterparty risk problem still exists and anyone would like to know if the company repays the bonds either in full or part (a transparent and auditable process). This can be verified if the exchange rate used by both parties is the same, but they could differ (even if slightly) as well. Is there a universal consensus on the exchange rate or would an oracle be consulted for the rate? This will be discussed at length in a later section of this chapter.

4.5 Design

As earlier stated, a Credit Event occurs when a party fails to fulfill the terms written in a loan contract. In other to implement the lending infrastructure, two approaches can be considered:

In one approach, a loan has a credit event object within itself where the credit event is a variable contained within the loan contract. One problem with this approach is that the protection contracts (Insurance, Collateral) would have to monitor the object on each loan deployed.

It would be interesting if the Credit Event object sits at its own address such that protection objects would not be worried about each loan that they insure individually. The protection objects would just have a global view of all credit events from a single address given specific loan identifier (such as the loan address). This approach uses a model called Interfaces. An interface specify that certain functions must exist. Interfaces are similar to

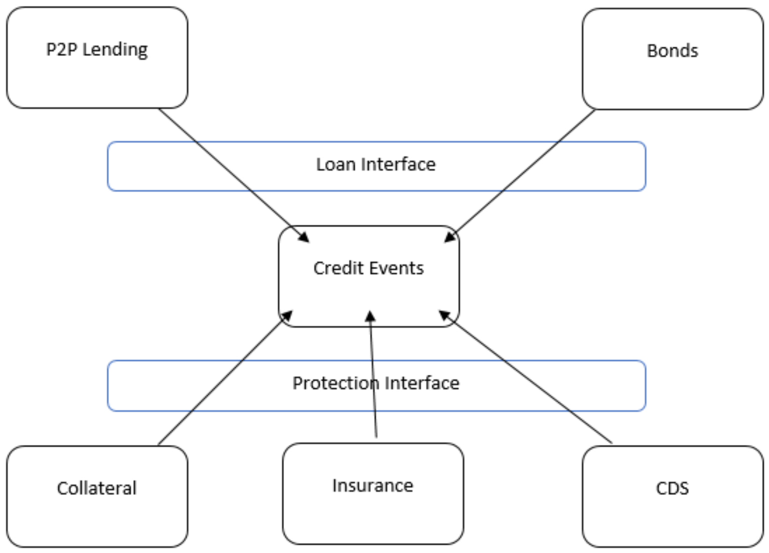


Figure 4.2: A lending infrastructure showing how the various loan and protection objects interface with the Credit event object

abstract concepts and therefore do not have any definition of functions contained within. An interface provides the user good guide as to how to implement the contract. It would force user to implement functions in a fixed way to ensure retrieval of desired information. This approach stimulates a table that contains all information regarding loan defaults visible to everyone.

In our work, we introduced two interfaces: a Loan interface and Protection Interface. The Loan interface forces any loan object that would like to interact with the Credit Event object to implement certain functions that would enable the interaction. This same concept applies for the Protection Interface in that protection object that would want to interact with the Credit Event object need to implement certain functions to be able to do so. These interfaces and their links to the loan objects are shown in Figure 4.2.

4.5.1 Loan objects

In this section, we walk through each loan object starting with a simple loan and ending with a complex loan. We also highlight the rationale behind every step taken.

Peer to Peer Lending

We start with a basic loan contract model constructed by a lender. The loan has parameters such as the address of the lender and debtor, the principal, start and end date and so on. The contract begins with a constructor function that initializes the mentioned variables and then a function that allows the lender to send the principal into the contract. The borrower comes along and triggers a function in the loan contract to retrieve the principal in the contract. At maturity, the borrower triggers a different function to pay back the principal with the corresponding interest. To ensure that only the stated or intended party run a function in a contract since all functions within a contract by default are public, the concept of Modifiers discussed in previous section was used, as shown in algorithm 4. The following are some issues we address in this basic contract.

If the borrower does not show up to retrieve the principal from the contract, the lender's money would remain the loan contract forever. To combat this, a kill function was implemented such that the lender can retrieve the money from the contract if the borrower never retrieved the money after a certain period of time.

Functions in an ethereum-based contract do not have the notion of order in the sense that the functions can be triggered by anyone in any order. Therefore, it is necessary to create a form of order, such as a state machine, that will control the flow of the code. For example, our state machine will ensure that the borrower can only run the function to retrieve the money only after the money resides in the contract.

It is needful to note that every Loan contract starts with this basic model described above. One crucial element missing from the basic loan contract is the implementation of what happens when a default occurs. If the borrower fails to pay after the due term, how could this action be made transparent to all user? The first approach discussed above was used in this next stage: the model was implemented in such a way that the default function can be triggered by any person watching or monitoring the loan if the borrower fails to pay after the due term. This default function when run, updates a default boolean variable from false to true. This way the loan moves into a defaulted state. This approach works best with a single loan and a single protection object watching the loan so that actions can be taken based on the state of the loan contract.

Bonds

A simple bond allows purchase of debt by any address with a defined interest to be paid at maturity. The contract starts with the creation of a token contract using the standard token available on ethereum github page.[1] This creates as many tokens as needed by the organization issuing the bonds. The contract for issuing the bonds is then created by the organization and the token are sent to the contract. The bond contract has a constructor that is used to set all variables (start date, end date, principal). A function is used to accept payment from investors where a token representing an amount is sent to the investors. The token is calculated as the value deposited over the price of the bond. An event is created that informs watchers of the contract of any bond sold. The bond is a Bearer bond in that the contract does not need to hold the address of a corresponding buyer to the number of tokens owned. The token can be transferred from one person to another without interacting with the bond contract. However, the interaction is performed with the standard token contract. To get paid at maturity, only the token needs to be considered irrespective of the bearer of the token. Defaults are implemented the same as in the P2P lending contract. The default function can be triggered by any person watching or monitoring the bond if

[1] http://github.com/ConsenSys/Tokens/blob/master/contracts/StandardToken.sol

the organization defaults on its payment after the due term. This default function when run, updates a default boolean variable from false to true. This way the bond moves into a defaulted state.

What would be interesting to have is an object that represent a global view of all defaults such that the protection object would not have to monitor each loan but would rather monitor a global object that would give it the desired information. To put it another way, instead of a protection object to establish a connection with several loan objects, it would establish a connection with a CreditEvent object and 'subscribe' to changes regarding the loans it covers. This offers a better transparency model.

So far, the model discussed implements a zero coupon payment where the principal and interest are paid once at the end of the loan term. In the later part of the thesis, we extended this model to include coupon payments.

4.5.2 Protection objects

As described earlier, protection against loan defaults can be performed on the blockchain. Those described were modeled around traditional protection mechanisms for off-blockchain loans. However if loans themselves are on the blockchain, it is only logical to implement protection mechanisms alongside them on the blockchain. One type of protection discussed was collateral.

Collateral

Two types of collateral are defined in this space—a token collateral and an ether collateral.

A token collateral contract implements the standard token contract found on the ethereum github page. Usually this token should represent an ether value assets such as a token from a DAO contract or loan contract or any contract with reputation in which the debtor must have invested in and received tokens. For the sake of this thesis, the standard token was used. This contract is created by a borrower of a loan contract as a collateral in the case of a default. The constructor function of the contract states the amount of token the borrower is willing to put up as collateral. Another function allows the borrower to instantiate all agreements with the lender. These agreements were not included in the constructor function to open the collateral function to any investor. If at the end of the term the debtor defaults, a function to get the token is executed by the investor. The function first checks the loan contract tied to it to determine if the loan is in a defaulted state after which it sends the token inside the contract to the investor. Finally, a kill function is implemented to ensure that the tokens do not remain in the contract forever if an investor never surfaces or a default never occurs.

An Ether collateral differs from the token collateral in terms of the actual instrument used as a collateral. In the case of an ether collateral, ethereum's internal currency is used as a form of collateral. All other functions within the ether contract are implemented in the same fashion.

Credit Default Swap

In simple terms, a Credit Default Swap (CDS) is an agreement between two parties (a seller and a buyer) where the CDS seller fulfils the debt of the CDS buyer if a credit default or event occurs on a debt held by the CDS buyer with a debtor. For example, let us suppose that Alice has an amount she wants to invest. She considers investing in bonds from Corporation A. She wants a corporation with a good rating as the money is towards her retirement, but say Corporation A has an average rating. What Alice can do is loan the money to Corporation A for which she gets an interest either monthly or yearly and then insure herself against a default (such as a late payment). A CDS is one type of insurance which can be taken against any derivative. To insure this loan, Alice can meet a CDS seller

Bob, who has a very high rating and pays a premium (say 1% of the interest Alice get from corporation A) to Bob. In return, Bob would insure the loan and since the Bob has a high rating, this automatically overshadows the medium or low rating that corporate B has making the package of the loan with the CDS highly rated. Because the CDS market is not regulated, Bob is not required to put up any collateral or declare his solvency to insure the loan. In the case of a default, the CDS buyer (Alice) would receive the face value of the loan from the CDS seller (Bob) and Bob would take over the loan form Alice becoming the new owner of the loan. Hence the name Credit Default Swap.

The complication with the CDS market comes from the fact that Bob would not only insure Alice's loan, but would also insure as many loan as possible without having to put up any collateral for any of them. The CDS seller (Bob) makes profit if Corporation A and similar corporations for which he insured do not default on their debt. In addition to this, anyone can buy a CDS for a loan in which they do not hold any loan instrument or have any direct insurable interest in the loan. This is done to speculate on credit events and are referred to as 'naked CDSs'.

Naked CDS's can result in a situation where there are more CDS contracts floating around the market than the actual debt being insured. It also complicates who takes over the loan if there is a default. The market addresses this using an auction. Briefly a credit event auction is used to settle large number of CDS contracts at once where the loan is traded for a value which is less than the face value of the loan. The buyer of this loan becomes the new loan owner. The price at which the loan is traded for is directly proportional to the credit worthiness of the Corporation bond been auctioned. More Information can be found here.[2]

Because there is no required declaring or auditing of transactions and the CDSs are not traded on lit exchanges, the lack of transparency in this market became a huge concern for

[2] http://www2.isda.org/

the regulators after the 2008 financial crises. Our aim in this section is to once again pull the transparency offered by the blockchain into the CDS market by creating a prototype of this market on the blockchain. Since all transactions within the blockchain are transparent to everyone, the buyers of CDS contracts can determine how many swaps are being insured by a particular swap seller (Bob) and also determine if it is safe to buy a CDS from the seller. Defaults of the swap seller on previous loans can be made visible on our Credit Score object. We would like to use this platform to introduce the concept of a CDS on the blockchain by implementing a simple CDS between two parties.

As said earlier, a more sophisticated model would be one in which the collateral contract would instead of considering each loan to determine if a default has occurred, would instead look at one contract that provides a global view of all loans it might be tied to.

4.5.3 CreditEvent

It would be simpler to implement a CreditEvent object on each loan (P2P or Bond) contract. One reason why we want to pull it out and make it an object of its own is to prevent redundancy in the use of code. This is a basic principle of object oriented programming. Another reason is to create a somewhat central place where all the loans can be looked at.

The simplest model of a CreditEvent object begins with a contract that holds all default variables such as defaulter's address, the lender's address and the defaulted amount. It implements a struct variable that is used to hold all the values pertaining to each loan. The contract implements the zero coupon payment model and hence has only one value for defaults. The value of the defaults could either be a string (yes or no) or a number (the amount defaulted). This contract has a constructor that is triggered by a loan contract. The major task of the constructor is to allocate memory for the loan that triggered it and set the necessary parameters (defaulter's address, the lender's address). An update function within the CreditEvent contract is triggered by loans to insert default value into the struct variable.

A defaultlist function acts as a getter function and returns all the values within the contract. This contract by itself performs no specific action beside receiving information from loans linked to it and acting as a global table visible to different protection objects and users.

In this simple model, the loan has a constructor that initializes all its parameters and triggers the CreditEvent function to insert arguments such as the lender's and debtor's address. A payback function contained within the loan is triggered by the debtor in other to pay back the principal and interest. It takes into factor the state of the contract as well as the maturity date of the loan. If the amount being paid by the debtor is less than the total amount (principal and interest), the amount is paid to the lender and the default is updates to the CreditEvent contract. A value of zero is updated to the CreditEvent contract if the amount being paid covers the total amount or in excess, in this case the rest of the money is returned to the debtor. A report function can be triggered by anyone watching the contract if the borrower defaults on its loan. This would set the loan to a default state such that anyone watching the loan can tell that the borrower defaulted on the loan.

Because most loans are usually implemented as a monthly coupon payment, a more classic loan would be one that implements a monthly coupon payment as opposed to the zero-coupon payment implemented above. In later section, we explain the implementation of the monthly coupon payment, noting the challenges encountered and the measures taken.

4.6 Deployment

This section discusses challenges encountered in the implementation of the project, as well as the steps taken to circumvent various issues. We include a few of our contracts in Appendix B for illustrative purposes, but due to space limitations, we do not include all of them.

4.6.1 Explaining the use of Oracles for exchange rates

It is not uncommon to encounter use cases that require a smart contract to trigger or change state in response to an event external to the blockchain. For example, an insurance contract might pay farmers based on the temperature and sunlight for a given period. A hypothetical smart contract might listen for any change in the weather, parse this information from an external source, and then trigger payments or other events based on this information.

As simple as this contract might sound, it is not possible to run contracts this way. This is because the blockchain follows a consensus-based model that ensures all data added to the blockchain has been processed and agreed upon by all the nodes on the network (or more strictly, a computational majority of nodes). Every miner executes transactions independently. Thus smart contracts must be completely deterministic with no room for discrepancies. Externally fetched data might differ between nodes, some nodes may not be able to access the data due to networking issues, and the amount of gas that should be consumed by the miner for spending time fetching the data is difficult to determine objectively. Thus the blockchain works in a sandbox model such that activities inside it are irrespective of activities outside.

In the case of our lending infrastructure, we want to implement a loan where the unit of account for the loan is based on the value of a fiat currency. The actual loan will be in Ether but the amount owed will be based on its current exchange rate with the underlying currency. This is because, as mentioned previously, the big problem with lending money in Ether is that price grows rapidly, which invariably means that if you take out a loan in ether, by the loan due term, the price of ether might have doubled.

As a result, we aim to get a feed into the smart contract that gives the current exchange rate of a fiat currency such that the amount at maturity can be accepted in Ether but denominated in the fiat currency.

For example, if a bond issuer needs a capital at 1 million CAD, he calculates the coupon payments in dollars and then converts it to ether and issues the bonds. For every coupon payment he makes, he calculates the equivalent in ether and pays it out. At the end of the term he pays the equivalent of 1 million dollars in ether. In nominal terms, the amount of ether being paid back might be more or less than the amount raised depending on whether it's value increased or decreased relative to the Canadian dollar. Bonds do not only offer an investment opportunity, but they allow investors to speculate or hedge on rates of inflation.

Since contracts cannot fetch external data, a rather sophisticated approach to achieving this is used by the community. Trusted parties known as oracles create a transaction that embeds the required data into the blockchain. This way, every node will get an identical copy of the data, giving every miner the same data for executing the functions in a smart contract. In short, an oracle sends data into the blockchain rather than having a smart contract try to fetch it from the external source.

For our implementation, we use Oraclize[3] an oracle to feed the price of the data into contracts. Using an oracle is not foolproof and we note a few challenges in using an oracle. The first challenge is that the price is needed at each execution of the contract. Another challenge is that in other to feed the current exchange value into the blockchain, a link to any exchange has to be manually inserted into the oracle's code. This in essence imply that if the link goes down, the oracle will not be able to provide the appropriate data into the blockchain to be used by the miners. A third challenge is that the source for which the oracle(Oraclize) gets it information from can lie about the prices and this can effect contracts. In this case we rely on the reputation of the source in other words their trustworthiness. It is worthy to note that all contracts using an oracle have this problem. Because much of everything in our world still works with a centralized systems, many of the decentralized applications still employ some degree of trust in a central party outside the blockchain

[3]https://github.com/oraclize

technology which in our case is the oracle. Now, a system shifts to either being classified as a decentralized or central system based on the degree of centralization employed. However, until every process within our society becomes decentralized, this would be inherent in most applications deployed on the blockchain.

4.6.2 Automatic payments of coupons via smart contracts

Consider a hypothetical smart bond that would be able to automatically transfer coupon payments at the specified times, with the hope that this would avoid the manual processes required and ensure that the organization cannot default. This bond is realizable on Ethereum and solves counterparty risk, but is not a sensible design.

One crucial fact often missed is that for a smart contract to make an automatic payment of some sort, the money needs to reside in the smart contract. Money can reside in smart contacts however if the funds used for coupon payments resides in the smart contract such that it never leaves the contract, then the payments would automatically be guaranteed. This model would be of no use to the bond issuer as it means that the funds cannot be used by the bond issuer for the intended purpose. But if the funds are used by the issuer as it normally should be, then there is no guarantee that the payment would be made.

Therefore, the idea of a guaranteed payment cannot be solved with the blockchain or emergence of smart contracts. This imitates a loan that is secured with a full collateral as discussed in previous section where the collateral is the same as the loan to be lent. In short, the smart bond would be pointless to the issuer. The purpose of a bond is to raise funds for a fruitful but slightly risky activity, such as erecting a new branch. If the money never leaves the smart contract, the bond issuer would have no way to use utilize the funds. This buttress the point stated earlier in previous section that the aim of this thesis is not to eliminate counterparty risk as every loan by default has counterparty risk but instead to make the process transparent as the link between risk and return cannot be solved currently with the blockchain.

Second, the term automatic used with smart contract is often over emphasized as functions cannot automatically perform an action except triggered by a subject. In this thesis, each loan has a function that should be triggered monthly to determine if there's a default by the borrower. This default function in the loan has to be triggered by a subject in other for the loan to move into a default state and to update the CreditEvent object. In this case, we had two choices. One choice is to use a system known as Ethereum Alarm Clock [4] to trigger the function monthly. Another choice is to allow the function to be triggered by one of the actors in the loan contract.

When a transaction is created, it is executed immediately and included into a block. What would be great is to have a transaction that can be executed at a later time following the occurrence of an event or after an elapsed time. The Ethereum Alarm Clock is a service that supports scheduling of transactions such that they can be executed at a later time on the Ethereum blockchain. This is done by providing all of the details for the transaction to be sent, an up-front payment for gas costs, which would allow your transaction to be executed on ones' behalf at a later time. Using the ethereum alarm clock involves heavy integration with the loan contract and would also involve a monthly payment for the service as well. Hence, there a question of if this service needs to be incorporated into the loan contract. Would it be possible for an actor in the loan contract to run the function monthly in other to avoid the heavy integration and cost of using the Ethereum alarm clock? Which actor in the loan contract would have a higher incentive to run the default function? All answers point towards the lender or investor of a loan contract. Due to the fact that the insurance or collateral can only be claimed after a default occurs, the lender in the contract would have more incentive to run the function every month at no cost. Hence, the second choice was implemented.

[4] http://www.ethereum-alarm-clock.com/

4.6.3 Implementing the monthly array object

In creating the concepts of months, to implement a monthly payment, we need the notion of time (now, block timestamp) or block interval (based on block number). Ethereum developers would warn against the use of block timestamp as it can always be manipulated by the miners. This is due to the fact that since it is a decentralized system, the clocks are not synchronized to the millisecond and therefore miners are allowed to have blocks that are either 900 millisecond behind or ahead. This is a challenge for applications such as prediction market, auctions and order book. In this scope of work, the two methods where tested to understand the impact of using either and the best method to use.

When using the block number, there is no clear or precise value to be used to represent a month. Since a block is always mined every 17 seconds on an average, this would mean a month can be represented using approximately 133,920 blocks. This is always an estimate and in reality, more blocks or less blocks would be mined before that period. Therefore, the concept of block number holds no true representation of the concept of months in reality. This was tested and confirmed. The other approach which uses block timestamp was implemented because our project is not time critical and therefore any manipulation performed by a miner would not break the contract. The only challenge here is that the block timestamp is recorded in seconds and this has to be converted into a month.

Moving further, we wanted to make it possible to ensure that if a month was missed (i.e either the Cash taker missed a payment or the cash giver failed to run the default month), anytime the function is executed it would check the current month of the loan contract against the array in the CreditEvent that holds the monthly defaults. If an element has not been inserted into previous months, the function inserted those elements before inserting the current value at the time the function was executed. Because, we used dynamic arrays to allow arbitrary number of months, we could not loop through the array to determine if an index had no element due to the fact that dynamic arrays only grow as elements are pushed into it. A simple solution was to check the length of the array in the CreditEvent against the current month in the loan contract. If the two values are the same, the array is up to date. If not, the missing elements are first populated before the current element for the current month is inserted.

4.6.4 Implementing the CreditEvent contract

In the CreditEvent, we needed each loan to have a separate container where each container holds the variables such as the address of the lender and borrower and another container within that holds the monthly payment for a given loan. There are different level that can be used to holds values of variables contained within the CreditEvent contract. The various combination are: a mapping contained within a struct, a struct contained within itself, an array within a struct.

According to solidity documentation [5], in order to restrict the size of the struct to a finite size, a struct is prevented from containing a member of its own type. However, the struct can itself be the value type of a mapping member. Following that, another way is to have a mapping to another struct outside that contain the monthly defaults. In solidity, mappings are like hash tables that are initialized virtually in a way that every possible key exists and those keys are mapped to a value that has an all zero byte-representation. However, it is not possible to get a list of all the keys of a mapping, or a list of all the values in the mapping. Due to the limitation of the mappings that prevents fetching the whole content of the variable at once at any given time and also the inability to loop through elements of the mapping which was needed in later part of the project, the best implementation was to have an array contained within the struct. This array holds the monthly defaults of a debtor for a specific loan.

Whichever way it is implemented, the container within cannot be visible within the

[5] https://media.readthedocs.org/pdf/solidity/latest/solidity.pdf

interface of the Ethereum wallet even if the outside container is given a public access mode. This is due to the fact that for example if you implement a struct inside another struct and on, eventually the interface would give up trying to display all the subviews within it. For debugging purposes, it is essential that the container within is visible. To view all different layers the getter function is used to return all views.

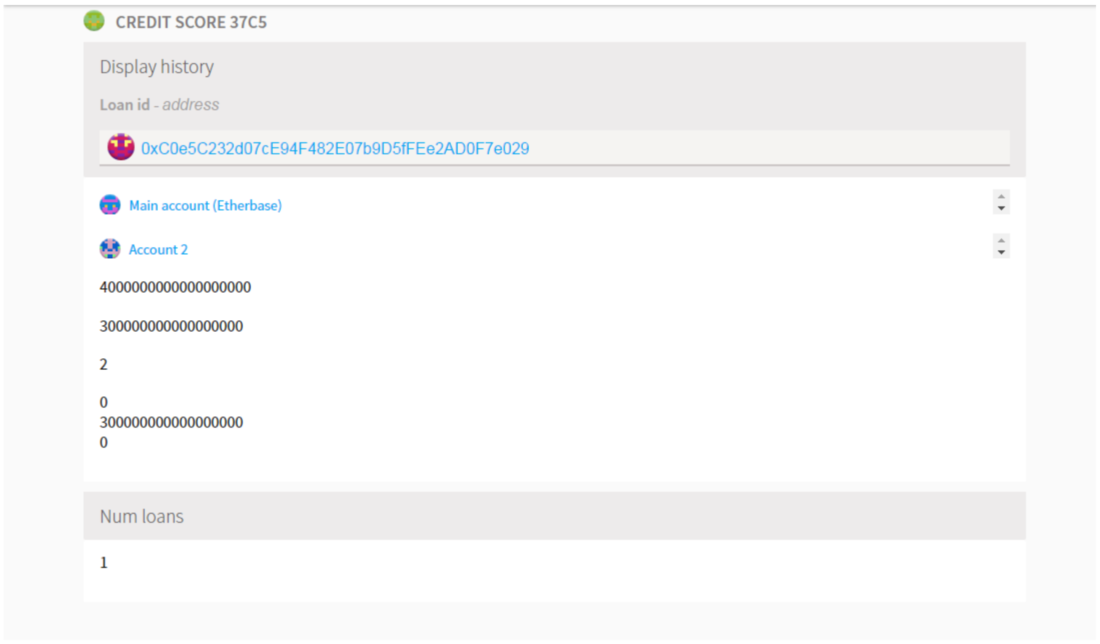
As discussed above, arrays and struct where used in the implementation of the CreditEvent contract. Arrays can be implemented in two ways- dynamic or static array. In this thesis, the dynamic array was used because we wanted the length of the array to be flexible which would be dependent on the number of months agreed upon in the loan contract. It is possible that the lender forgets to run the default function for a particular time. Therefore, when the function is run at any given time, it is only wise for the function to check if some months have been skipped by the lender. In other to do this, it tries to determine the current month in which the loan is. Since we are dealing with dynamic arrays, it is not possible to check the array in the struct for an empty value. Instead the current month in which the loan reside is checked against the current length of the array that resides in the CreditEvent contract. This way the function would be able to determine if values where entered for the previous months and populate appropriately.

In other to uniquely identify loans in the CreditEvent contract, when a loan calls the CreditEvent contract to pass in the initial parameters, a loan id number is created by the CreditEvent contract. This loan id number can be used by a protection object to monitor a loan. Using a loan id number creates an extra variable that floats around the contract that might not necessary be needed. A better approach is to use the loan address as a unique identifier. This way the protection object do not need to keep the loan id number of every loan they monitor as the address of the loan by itself serves as a unique identifier. This however, is not a hard rule as either a loan ID number or address can be used to uniquely identify a loan without causing any mishap in general. Even in situations where two loans are created at the same time, the id of the loans is set by the miner in the order in which they are place within the block. Figure 4.3 shows the interface of the Ethereum wallet for the CreditEvent object. It contains the parameters for identifying each loan on the CreditEvent object. The loan address is used to retrieve this information as seen in the figure. The months which have no default are represented with zero and 300000000000000000 wei(0.3 ether) is the defaulted amount for the second month. To pay out this default, any protection object would just need to fetch the value from the CreditEvent object.

The classic approach discussed in implementing a CreditEvent object was difficult to actualize for the bond contracts since the interfaces for both differ. Where the loan would call a function to create a container containing the address of the cash taker, cash giver and an array for the monthly payment, the bond on the other hand would require a function that would create a container for the bond and then create sub containers representing the different investors of the bond alongside an array for their monthly payment. Thus, a single interface which in this case means a single CreditEvent might not be feasible in this case. One approach to circumventing this problem is to create different containers for the different investors as one would for a Peer to peer lending contract. For example, instead of the function called by the bond creating just a single view and adding all the investors under that view, the function can be called as many times up to the number of the investors for that given bond.

4.6.5 Implementing a Credit Default Swap

In the implementation of the credit default swap, it is ironic to say that the CDS contract operates as both a loan and a protection object at the same time. The CDS contract is drawn up by the CDS seller who initializes agreed upon facts such as the CDS buyer, amount to be insured, premium, among others. Because the CDS contract contains a loan, during the first payment of the premium by the CDS buyer, the function is triggered that allocates


CREDIT SCORE 37C5

Display history

Loan id - *address*

0xC0e5C232d07cE94F482E07b9D5fFEe2AD0F7e029

Main account (Etherbase)

Account 2

```
4000000000000000000

300000000000000000

2

0
300000000000000000
0
```

Num loans

1

Figure 4.3: A snap shot of the CreditEvent on the Ethereum wallet

space in the CreditEvent object to hold information regarding the standings of payments made to the CDS buyer.

Now to the heart of the matter, if a default occurs on a loan that has been insured with a CDS, the default function which would be run by the CDS buyer (the CDS buyer has a higher stake and more incentive to run the function) would update the CreditEvent object with the balance of the loan to be paid as opposed to the default for that month as in other kind of loan deployed previously. This is because when a CDS occur, the rest of the debt is paid to the CDS buyer and the CDS seller takes over the loan (this is where the swap occurs). The idea behind this is that we wanted the CDS contract to fetch the balance of the debt directly from the CreditEvent object just as the Collateral object gets the default for the month from the CreditEvent object and pays out to the Lender (Cashgiver). This way the amount to be paid cannot be manipulated by either the CDS seller or anyone and the payment can be made automatically when triggered.

When the payment is made to the CDS buyer, a change of ownership occurs. Now, this could be implemented in two ways. One way is to have a new contract created for the change of ownership where the CDS seller becomes the Lender in the loan contract. This would create a new contract which might be oblivious to others as it would have a new address which has no relation to the old address. The other way, which we implemented, is to have the same loan contract implemented for the CDS change ownership name. This way the new owner (CDS seller) is tied to the loan contract and anyone who had the address of or watching the CDS loan would be aware that a credit swap just occurred. The change of ownership also must be reflected on the CreditEvent object.

Due to the nature of CDS the contract terminates (that seizes to exist) if the CDS buyer defaults on the premium at any point in time. Therefore, it would be in the interest of the CDS buyer to pay the monthly premium. The rest of the functions within the CDS contract works in the same manner as explained in previous sections.

General challenges encountered during the course of this project resulted due to the fact that the Ethereum is at its conception stages. This results in lots of changes made daily to the structure and semantic of the programming languages such as solidity used in this project. Most codes that were previous written and worked correctly began to throw warnings and error when complied with the latest compilers mostly due to change to keywords used, structure and semantic of the new compiler. This causes so much delays and lags having to debug the code in other to use it in the current context needed. This challenge is escalated when libraries used in this project suffer the same faith where most of them are not maintained by the authors resulting to having to painfully manually scan through the libraries to debug the errors such as the Oraclized API.

4.7 Testing

The Ethereum Virtual Machine has two network—the Main Network and the Test Network. The Main network is where real world smart contracts reside. As a result of the financial cost that would be required for paying for gas, it is impractical to deploy the experiment on the main Ethereum network while the contract is under development. It would also be unwise to deploy a half-baked contract on the main network. One response is to create a private Ethereum network to act as a test bed, while the other option would be to use Ethereum test network. In this project, we use the Ethereum test network as it is deployed exactly as the main network is (decentralized with a peer-to-peer network) but doesn't use real ether (it uses test ether that can be obtained easily for free).

Also in order to test our system for known security bugs, we use a symbolic execution tool developed by [29] called OYENTE.[6] The tool was deployed on the Ethereum blockchain and has been proved in successfully identifying critical security vulnerability, such as a famous incident called the DAO vulnerability.

4.8 Evaluation

In this section, we evalute our lending infrastructure based on the following requirements that were set out during the design of the project.

4.8.1 Transparency

Transparency is a main factor in the design of this system. The aim of the project is to leverage the transparency provided by the blockchain technology. Our system is transparent in the following ways. All contracts are made visible to everyone within the network and anyone who wishes to participate in the lending infrastructure can have insight into what exactly the system does and how it claims to do what it does. Once a system has been deployed to the network, the underlying code cannot be altered. This imbibes trust into the system. Second, all transactions executed on Ethereum network are made visible to

[6]https://github.com/ethereum/oyente

everyone and with the Event keyword we use in Solidity, subscribers can be notified of any changes made to the system enabling real time monitoring of events within the contracts. Last and most importantly, the system has been designed with the sole aim of making defaults of loan contracts visible to all. With the implementation of the CreditEvent object, all defaults for any loan connected to the CreditEvent object is located at an address (the address of the CreditEvent object) which allows for easy tracking and monitoring by any node including protection nodes (collaterals and insurance). With transparency incorporated into the lending system, more users can be confident in exploring lending within the blockchain realm.

4.8.2 Reliability

An aim of this project was to ensure the availability of the smart contract at any given point in time. By default, the Ethereum Virtual Machine is a decentralized system that eliminates any single point of failure common in centralized systems, and hence provides reliability for all smart contracts contained within.

4.8.3 Security Bugs

As with any new programming language, security vulnerabilities are inevitable and Solidity is not an exception. Ensuring that smart contracts are safe and free of bugs is of utmost concern as these smart contracts hold virtual currencies that have real world value. Any smart contract that holds money is open to attack. As stated by [15], programming smart contracts would require an economic perspective that is unusual for traditional programmers because money is directly incorporated into the contracts. Contracts must be coded to guarantee fairness even when counterparties attempt to circumvent or cheat in arbitrary ways that maximize their economic gains. This section will provide some details on known security vulnerabilities and remediation implemented during the course of this project. Thorough systemizations of Solidty (and Ethereum) vulnerabilities can be found in the literature [15], [29], [39]. In addition to careful design, we validated our code using OYENTE [29].

Re-Entrancy

The first instance of this bug was discovered during an attack on The Decentralized Autonomous Organization (DAO). The aim of the DAO was to eliminate the need of a third-party organization in handling fund raising of a new idea or an organization. This was actualized through crowdfunding where tokens were given to the investors of the company. However, an attacker was able to part with one third of the money in the DAO contract worth approximately 86 million USD (as of the time of the attack) due to an issue in the splitDAO function that was responsible in managing and funding new child DAOs or projects. Below is a brief detail of the attack.

For any contract A(caller) and B(called), any interaction or transfer of funds between the contracts hands control of the caller contract over to the called contract. Once control has been handed to the called contract (contract B), B can repeatedly call into A before the end of the interaction.

The contract above acts like a bank that allow users to deposit money, keep a record of each balances and withdraw their money at will. The contract can be exploited within the withdraw function. This is due to the fact that the update of the users balances after the withdrawal occurs after money has been withdrawn. This can be exploited by an attacker in that because a call() automatically triggers the fallback function of any contract that it sends money to, the fall back function could contain the withdraw function. And this would execute repeatedly until the gas limit is reached or the money in the contract is out. All this while without jumping to the line that updates the balance of the user. Hence, the term Re-entrancy.

Algorithm 8 Re-Entrancy

```
pragma solidity 0.4.10;
contract Bank {
    mapping (address => uint) public balances;
     function Bank ()
     {
      address owner = msg.sender;
     }
     function payable;
     {
      balances[msg.sender] = msg.value;
     }
     function withdraw()
     if (!msg.sender.call.value(balances[msg.sender]))
     {
       throw;
     }
     balances[msg.sender] = 0;
     }
}
```

Mishandled Exception

In Ethereum, special functions such as send, call or any other function being called by a contract or external address can misbehave if any exception is not properly handled. Exception raised such as exceeding call stack limit or insufficient gas in the called contract, terminates the current execution, reverts its state, throws an exception or returns false. However, depending on what call was made, the exception in the called contract may not propagate to the caller contract. For example, in EVM the maximum call stack is 1024. If this stack is reached by an external function, it causes it to fail and an exception is thrown.

However, in the case of a send(), an exception is not thrown and this can be exploited by an attacker. This is generally referred to as the Call-stack depth. They can avoid a payment needed to be made before accessing functions in the contract by forcing the contract to reach its maximum depth that returns false rather than throw an exception. If the false exception is not properly handled, then the attacker would have access to functions or variables it otherwise should not have access to. Inconsistent exception propagation behavior as seen, results in many cases where exceptions are not properly handled. It is best practice to ensure that if a call is made via the send function, the caller contract should always check the return value to verify the call was executed properly before proceeding to other functions.

Timestamp Dependence

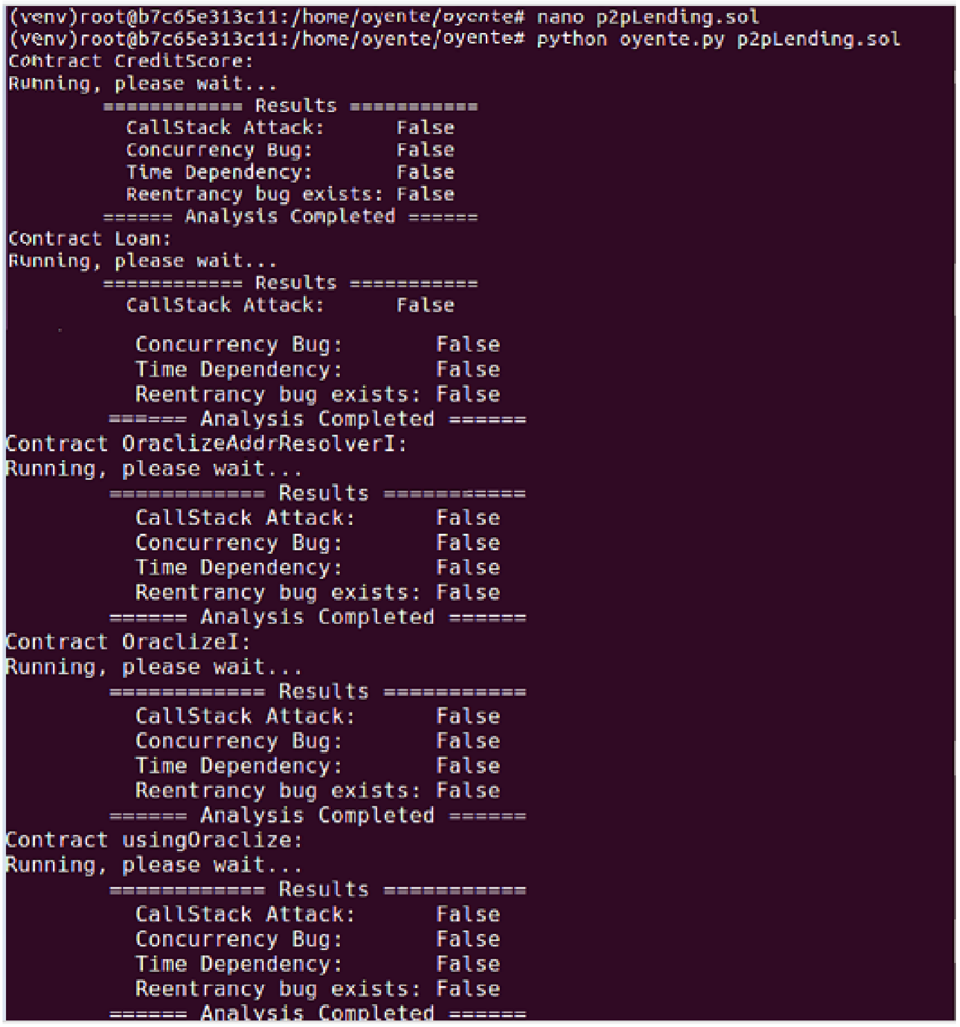
Many applications are dependent on some form of time factor in other to determine what executions to be performed or what actions are allowed in the current state of the system. In Ethereum, time constraints are implemented using block timestamp. In other to ensure time synchronization by all miners and factoring lags in various miners' clock, an upper bound on time synchronization by 900 seconds is agreed upon by all the miners. Typically, all transactions inside a block share the same timestamp. This ensures a consensus on the state of the contract after the execution. However, this may also expose the contract to being exploited by miners. Indeed, if the miner has a stake in a contract to be mined, he could gain an advantage over every other stakeholder by choosing a suitable timestamp for the block being mined where the timestamp is within a certain degree of randomness, that is a tolerance of 900 seconds.

Transaction-Ordering Dependence

This occurs when the order of the transaction can be manipulated to gain an advantage. An example is a contract where the price of an item is not fixed and liable to change let's say an auction. It is possible that the current price of the item is $20 and a user who is willing to buy the item at that price sends a transaction. We are going to assume the seller happens to see the transaction as if passes through the network. Now the seller can update the price of the item from $20 to $40 and if his transaction appears first in any miners' block, the buyer would be charged $40 for the item as opposed to $20 which he saw. Bare in mind that the seller could also be a miner within the network.

Our Contracts

As discussed above, we made our contract resilient to the re-entrancy bug by ensuring that all checks are performed before transfers (such as, does the sender have enough ether) and also ensuring that state variables are changed before transfers. Mishandled Exceptions have the potential to allow unauthorized access to functions or result in Denial of Service attacks on individual smart contracts. We handle this in our contracts with the use of modifier functions that act as an access control mechanism. This allows only authorized users to access functions and also sanitizes inputs to reduce the likelihood of exceptions. Transaction-Ordering Dependence and Timestamp Dependence attacks do not break our contract due to the nature of our project. Although, timestamps (as oppose to blocknumber) are implemented in our project, our contract is not time dependent and any modification of the time by factor of 900 seconds by the miner will not break the contract. Last, the price for a bond in our system is fixed by the bond issuer and cannot be changes after deployment. Therefore, the contract is not susceptible to a transaction ordering dependency attack. Figure 4.4 and Figure 4.5 shows the results of Smart Contract analysis tool on our smart contracts deployed for the peer to peer lending and bond contract respectively. The credit default contract has the same output and was not inserted into this write up for no other reason than a repetitive display and space. The various APIs used by both contracts were analyzed together simulating the exact same way it would be deployed. The results are also shown in the figures.



```
(venv)root@b7c65e313c11:/home/oyente/oyente# nano p2pLending.sol
(venv)root@b7c65e313c11:/home/oyente/oyente# python oyente.py p2pLending.sol
Contract CreditScore:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
Contract Loan:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
Contract OraclizeAddrResolverI:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
Contract OraclizeI:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
Contract usingOraclize:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
```

Figure 4.4: Results of Smart Contract analysis tool called Oyente [29] to find security bugs on the Peer to peer lending smart contract

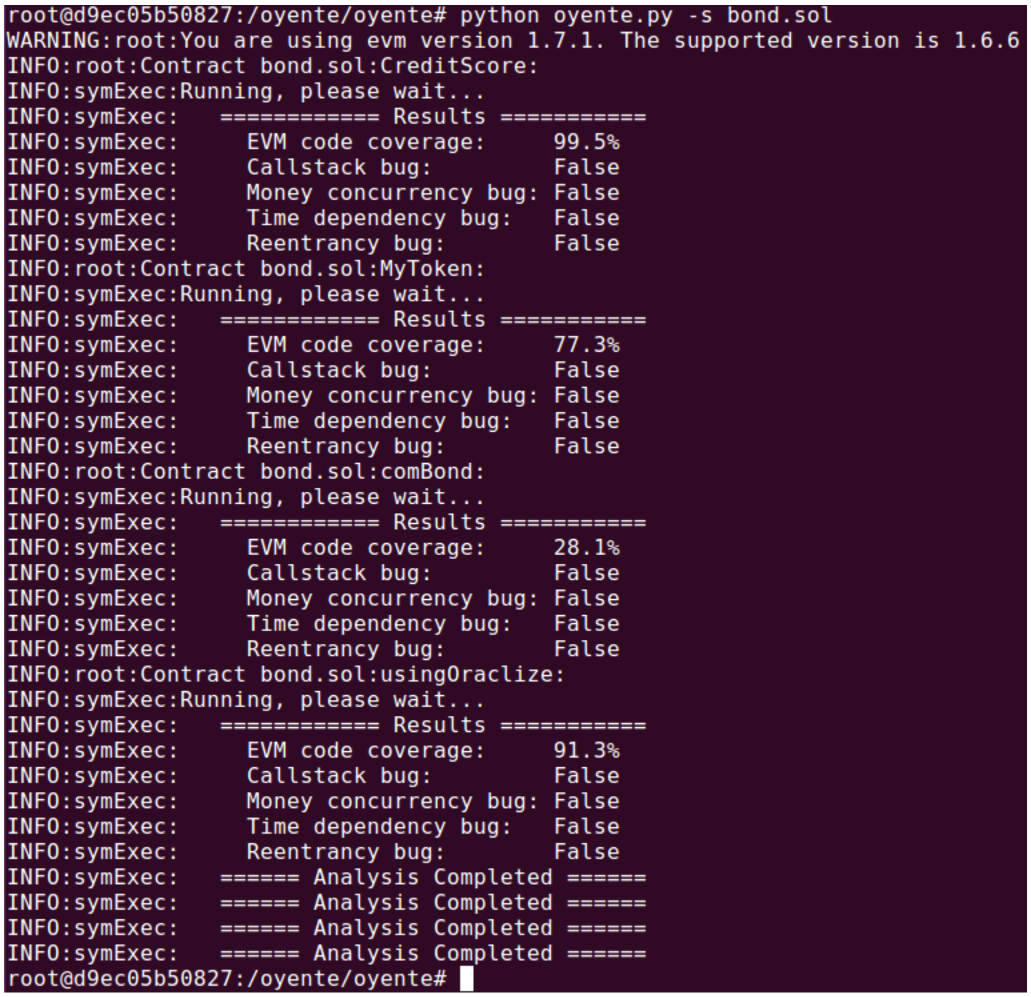

```
root@d9ec05b50827:/oyente/oyente# python oyente.py -s bond.sol
WARNING:root:You are using evm version 1.7.1. The supported version is 1.6.6
INFO:root:Contract bond.sol:CreditScore:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     99.5%
INFO:symExec:     Callstack bug:         False
INFO:symExec:     Money concurrency bug: False
INFO:symExec:     Time dependency bug:   False
INFO:symExec:     Reentrancy bug:        False
INFO:root:Contract bond.sol:MyToken:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     77.3%
INFO:symExec:     Callstack bug:         False
INFO:symExec:     Money concurrency bug: False
INFO:symExec:     Time dependency bug:   False
INFO:symExec:     Reentrancy bug:        False
INFO:root:Contract bond.sol:comBond:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     28.1%
INFO:symExec:     Callstack bug:         False
INFO:symExec:     Money concurrency bug: False
INFO:symExec:     Time dependency bug:   False
INFO:symExec:     Reentrancy bug:        False
INFO:root:Contract bond.sol:usingOraclize:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     91.3%
INFO:symExec:     Callstack bug:         False
INFO:symExec:     Money concurrency bug: False
INFO:symExec:     Time dependency bug:   False
INFO:symExec:     Reentrancy bug:        False
INFO:symExec:   ====== Analysis Completed ======
INFO:symExec:   ====== Analysis Completed ======
INFO:symExec:   ====== Analysis Completed ======
INFO:symExec:   ====== Analysis Completed ======
root@d9ec05b50827:/oyente/oyente# 
```

Figure 4.5: Results of Smart Contract analysis tool called Oyente [29] to find security bugs on the Bond smart contract

4.8.4 Cost

The definition of Ethereum gas and gas price are found in chapter 2 . In this section we would analyze the gas cost of using our contracts. As of this writing, the current price per gas is 21 gwei (0.000000021 Ether) while the current price of 1 ether = $277.78. For any contract, the gas cost = gas * gas price. As of this writing, it is useful to note that any transfer of ether from one account to another has a gas of 21,000, a gas cost of 0.00044 Ether resulting to $0.12 USD. Table 4.1 to Table 4.6 represents the cost of running each smart contract and its functions contained therein on the Ethereum Virtual Machine.

In Table 4.2 and Table 4.4, the cost of deploying the p2p lending contract and the Bond contract is roughly about $13.00 respectively. This is due to the API's called by those contracts, the more API's a contract import the more the code needed to be executed by the miners and the higher the gas consumption. In particular, the high gas consumption is attributed to the Oraclized API. However, once deployed, the cost of running the rest of the function inside the contract is less than $3.00.

Table 4.1: Cost of running the basic Libraries contract needed

Contracts	Gas	Gas Cost(Ether)	Gas cost (USD)
Tokens	857,106	0.018	$5.00
Token Transfer	51,501	0.001	$0.30
Oraclized	154,711	0.003	$0.90
Credit Score	462,453	0.010	$2.70

Table 4.2: Cost of running the Peer to Peer Lending contract

Contracts	Gas	Gas Cost(Ether)	Gas cost (USD)
P2P Lending	2,198,423	0.046	$12.82
Receive money	474,112	0.009	$2.77
Payback	105,827	0.002	$0.62
Report default	60,605	0.001	$0.35
Kill	25,098	0.001	$0.12

It is useful to note that the kill function in Table 4.2 can only be executed before the borrower or cash taker receives the money just in case the borrower never shows up. This is to prevent the money from been locked in the loan contract forever.

Table 4.3: Cost of running the Bond contract

Contracts	Gas	Gas Cost(Ether)	Gas cost (USD)
Bond	2,229,084	0.047	$13.00
Purchase bond	231,397	0.005	$1.35
Withdraw	292,787	0.006	$1.71
Repay	415,213	0.009	$2.42
Report Default	55,798	0.001	$0.33

Table 4.4: Cost of running the Collateral contract

Contracts	Gas	Gas Cost(Ether)	Gas cost (USD)
Collateral	442,035	0.009	$2.58
serve	204,509	0.004	$1.20
Get ownership	312,667	0.007	$1.82
Cancel	27,664	0.001	$0.16

Table 4.5: Cost of running the Credit default swap loan contract

Contracts	Gas	Gas Cost(Ether)	Gas cost (USD)
P2P Lending	2,198,423	0.046	$12.82
Receive money	474,112	0.009	$2.77
Payback	105,827	0.002	$0.62
Report default	60,605	0.001	$0.35
Change Ownership	30,605	0.001	$0.35
Kill	25,098	0.001	$0.12

Table 4.6: Cost of running the Credit default swap contract

Contracts	Gas	Gas Cost(Ether)	Gas cost (USD)
CDSContract	452,035	0.009	$2.58
montlypremium	204,509	0.004	$1.20
Report default	61,709	0.001	$0.16
kill	27,664	0.001	$0.16

Chapter 5

Concluding Remarks

The aim of this project was to introduce some new applications of blockchain technology aside from the creation of new cryptocurrencies. First, a survey of distributed ledgers against various highlighted criteria was conducted. Each of the discussed ledgers were matched against each criterion, providing a wide landscape that allows anyone to select a ledger based on the criteria needed in the application.

Second, a study of the integration of verifiable voting with blockchain technology was presented to show the potential of the integration as well as areas where further work is needed to actualize the full benefits offered by the blockchain technology.

Finally, a decentralized lending infrastructure system was implemented that harnesses the transparency of the blockchain to provide a system that is transparent and requires little interventions from intermediaries.

5.1 Limitations and Future Work

We cannot emphasize enough that contrary to the popular belief, smart contracts cannot automatically perform functions on a schedule or when external events occur (or internal events that do not explicitly trigger the contract). To address this, there is a concept known as Ethereum Alarm Clock that is designed to trigger events at specified times. It gives rewards to any node that does the triggering as an incentive to run the function. As explained earlier, due to the cost involved with using the Ethereum Alarm Clock, we decided to rest the burden of ensuring that the default function is executed monthly on the lender if neither the debtor nor any other node triggers the contract. However future work could explore the use of an alarm clock feature.

An inherent limitation of our approach is counterparty risk. Contrary to popular belief, blockchains do not solve the problem of counterparty risk associated with any kind of loan. Except a fully collateralized loan, all loans are susceptible to a form of risk. As earlier mentioned, we aim to instead make the implementation and use of loans transparent. Therefore, our lending infrastructure is not foolproof to counterparty risk but offers mitigation strategies.

In the implementation of the bond contract, our main aim was to implement a notion referred to as a bearer token. A bearer token is one that has the same property as cash in the sense that anybody can give cash to anyone and once you have the cash you can purchase anything with it without a reason to identify the cash owner. Although we were able to achieve this to some degree, it was only achieved within the bond contract but the token contract in which the token originates from before it was transferred into the bond contract still keeps a record of the token associated with the new address of the owner and any transfer of the token. Future work could explore options for making bearer bonds not rely on ledgers.

In the implementation of Oraclized, we could only embed one website to retrieve the current exchange rate with a weak assumption that the site would remain active. But this website might no longer be available in the future and the oracle would not be able to retrieve the needed data. A future work would be geared towards an oracle that allows an implementation of more than one website to retrieve its data site such that it removes a single point of failure.

Also due to the generic implementation of the Oraclized API to accommodate various uses cases and the proof performed to verify the accuracy of the Oraclized object, the source code is quite lengthy. This in turn increases the gas needed to execute each contract that utilizes the Oraclized API. Therefore, future work could be an implementation of an oracle designed solely for the lending infrastructure. This would in turn reduce the gas cost and reduce the price of executing the contracts.

The CDS implemented for this project was designed for insuring a P2P loan. However, this implementation can be easily integrated with the bond contract or any other financial derivative. Future work in this regard would be to implement a Credit Event Auction on the blockchain where the loan can easily be transferred at the correct price, reducing costs for clearing and settlement as these would all happen automatically by the smart contract when triggered.

Finally, it would be interesting to study floating payment bonds. Floating payments are another type of interest payment where floating rates are reset at different intervals depending on the market value. The focus of the project was on zero/regular coupon payments.

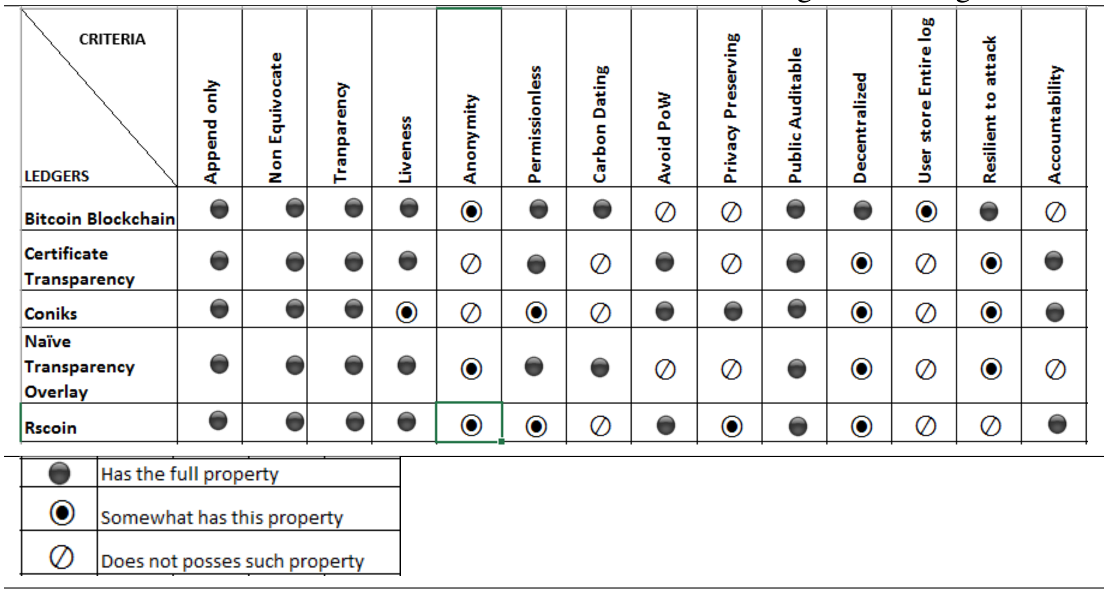
Appendix A

Evaluation Framework

A.1 Bitcoin/Ethereum Blockchain

Bitcoin is an open sourced Cryptocurrency and payment system based on cryptographic proof rather than trust. Bitcoin Blockchain simply put, is a public ledger of all Bitcoin transaction since the conception of the cryptocurrency. It is a decentralized databased that is shared by all users participating in the system. Upon joining the network, each client connected to the bitcoin network receives a copy of the blockchain and subsequently updates it when connected to the internet. The integrity of the blockchain is maintained with Proof of Work (PoW) with the clause that majority of the hashing power is contributed by honest users. Bitcoin functions in the following manner. Users create public and private key pairs and are identified by their public key pair known as addresses. This address is within the Bitcoin network and have some amount of bitcoin store in them. Sending a bitcoin to another user is a form of transferring ownership of bitcoin to another user which involves the sender creating a transaction with the desired amount as input and the receivers address as an output. The transaction is cryptographically signed by the sender to ensure that the sender authorized this transfer. This transaction is then broadcasted to the bitcoin network where it reaches a miner who inserts it into his block of transactions to be mined.

Table A.1: An evaluation framework for distributed ledger technologies.



CRITERIA / LEDGERS	Append only	Non Equivocate	Tranparency	Liveness	Anonymity	Permissionless	Carbon Dating	Avoid PoW	Privacy Preserving	Public Auditable	Decentralized	User store Entire log	Resilient to attack	Accountability
Bitcoin Blockchain	●	●	●	●	◉	●	●	⊘	⊘	●	●	◉	●	⊘
Certificate Transparency	●	●	●	●	⊘	●	⊘	●	⊘	●	◉	⊘	◉	●
Coniks	●	●	●	◉	⊘	◉	⊘	●	●	●	◉	⊘	◉	●
Naïve Transparency Overlay	●	●	●	●	◉	●	●	⊘	⊘	●	◉	⊘	◉	⊘
Rscoin	●	●	●	●	◉	◉	⊘	●	◉	●	◉	⊘	⊘	●

●	Has the full property
◉	Somewhat has this property
⊘	Does not posses such property

When mined, the miner broadcast the block, the block is accepted by the network and hence included in the blockchain where the receiver can confirm the transaction and claim ownership of the money.

(1) Append only: Since it follows the Merkel tree data structure where logs are chained to each other to obtain a single Merkel root, any change in previous record will result to a change in the Merkel root which would be detected during verification by any number of nodes.

(2) Non-Equivocate: Bitcoin blockchain ensures consistency in the log across users of the system because for one, it is not a centralized system as the log is maintained by very participant in the system. And second if an attacker attempts to present two conflicting views to different users, the attacker would have to compromise the whole system that is create a fork which very user would be aware as blocks are being propagated within the network.

(3) Transparency: All transactions within the blockchain are visible to all users within

the system.

(4) Liveness: Due to its decentralized nature, transactions when created are broadcasted to the network where any miner can include such transactions to his block. Hence since the miners are operate individually under no centralized system, a given transaction is bound to be eventually included in a miners block and invariably in the blockchain.

(5) Anonymity: There exist a kind of anonymity known as pseudo-anonymity where public keys are used to identity users within the system but this identity only exist within the bitcoin network and therefore are not tied to their real world identity. Therefore, transactions are created, mined and validated pseudo-anonymously.

(6) Permission to join: The system is open to everyone and any one can send, receive and mine transactions with the network.

(7) Carbon Dating: Updating a block into the Bitcoin blockchain requires solving a puzzle (PoW), hence any log that was inserted x blocks from the tail of the blockchain was inserted x puzzle ago. As a result, it would take a great deal of time to compute x puzzle thereby making the message accumulate work over time much like how a fossil accumulate carbon over time.

(8) Avoids Proof of Work (PoW): Bitcoin blockchain is centered around PoW due to the following reason: first, since its not a centralized system, consensus has to be reached on the content of the block as well as the ordering of the block. Second, coins need to be minted and distributed within the system. Finally, for rate limiting the time for inserting a new block into the blockchain.

(9) Privacy Preserving: All transaction within the system are stored in plaintext.

(10) Publicly Audible: Since the ledger is public and no trust is assumed, its open to be audited by the users to ensure there are no double spends and all transactions contained within the blockchain are valid.

(11) Decentralized: It is a decentralized system in that miners who are responsible for adding blocks to the blockchain are not named trusted entity but basically any user with the system. All users within the system can be miners who are responsible for adding blocks to the blockchain.

(12) Do not store entire log: As a full node, the entire blockchain is stored by the client for verification purpose. Half nodes do not store the blockchain but perform verification through Simple Payment Verification (SPV) system.

(13) Resilient to attack: Since its a decentralized system with the condition that majority of the users are honest, an attacker would have to gain more than 50% of the hashing power in the network to either perform a double spend attack or modify blocks. Also for an attacker to modify a log embedded X puzzle ago, the attacker would have to compute a proof of work from the X block to the last block at the tail end. Hence the further the block is from the tail end, the harder it gets for the attacker to modify.

(14) Accountability: Since the users with the network are pseudo-anonymous, no one can be held responsible for any bad behavior. Although if a miner should post a block that contains invalid transactions, the block would be rejected.

A.2 CONIKS

CONIKS proposes to eliminate the need for centralized directory of public keys maintained by service providers by implementing a transparency log that solves challenges specific to key verification for end users. CONIKS retains the basic design structure of service providers issuance of authoritative name to-key bindings within their namespaces, while ensuring that users can repeatedly monitoring the consistency of their bindings thereby ensuring that any party A that wishes to communicate with party B can be confident that he has in possession the right domain name and corresponding public key of B. They aim to achieve two goals- Efficient monitoring and Privacy-preserving key Directories. Efficient monitoring of users binding can be achieved using a Merkel prefix tree where the root of the tree can be used to guarantee the consistency of a user binding in the directory. This enables users client to monitor their on key binding without having to rely on third party monitors. Also when a user views the entire log or an authentication path, the viewer become aware of users who have been issued certificates as well as their key data. CONIKS addresses this with a Privacy-preserving key Directories where response for domain name query leaks no information about other users existence in the directory. The participants in this system includes: Identity provider, Client software, auditors and users. The identity providers manage the CONIKS server and distinct namespaces with their own set of name to key binding. The Client software is hosted on the users trusted machine and it monitors consistency of the users name-key binding. The auditors tail the series of signed chained snapshots to ensure that identity providers are not equivocating and ensure global consistency by gossiping with one another. The users are participants in the network. Every version of the directory (implemented as a Merkel prefix tree) as well as every generated Signed Tree Root are persisted in a MySQL database. This log has the following characteristics.

(1) Non Equivocate: Any attempt by an identity provider to present two conflicting views of the name-to-key bindings in the namespace to different parties would be detected as the implementation of CONIKS allows auditors to communicate signed chained snapshot of logs issued by the CONIKS identity provider to each other. But this security model works under the assumptions that the all auditors act honestly and do

not collude with one another or the malicious log server.

(2) Transparency: Any alteration to the log is visible to all users within the network.

(3) Append only: The log structure follows the Merkel tree data structure where logs are chained to each other to obtain a single Merkel root, any change in previous record will result to a change in the Merkel root which would be detected during versification.

(4) Liveness: One fact that is certain is that once an Identity provider has issued signed temporary binding then the user binding is sure to be included in the log. However, it is possible that an identity provider can deny a users request for registration of a domain name. Another important denial of service could be the identity provider can refuse to respond to a query about individual binding to prevent detection of its malicious behavior. They offer a solution where other identity providers can proxy the users request which is still flawed as an identity provider can ignore all request for a domain name. This can somewhat be addressed if there are multiple identity provider for a service provider but this comes with its own disadvantages.

(5) Permission to join: The log is open to all users within the system and no permissions are required to query the log servers.

(6) Carbon-dated Messages: It does not have the notion of carbon dated messages as none of the users are concerned by how long a particular binding has been in existence rather their concern is more on if the user binding presented is accurate. Also the PoW needed to enforce the carbon-dated property is not used in this system

(7) Privacy-Preserving: CONIKS hides the number of users and key data mapped to each domain name to prevent adversary from learning anything associated to a public key. The goal here is to ensure that each authentication path reveals nothing about

other users present in the log. This is achieved by using Verifiable Unpredictable Function to transform domain name into what is referred to as private index. Also a cryptographic commitment to each users key data is also stored at the private index to ensure that it is impossible to check if a users key data exist on the log due to some known value.

(8) Anonymity: The users of the systems are known entities and the CONIKS server can tell which two entities are communicating due to the fact that for one party to communicate to another, one would have to request for the public key of the second party although the server cannot see the content of the message during the communication.

(9) Accountability: CONIKS server supports accountability in that the log server gives a receipt for any new or changes to a name-to-key binding of a user. The receipts which is a signed temporary binding consisting of the signed tree root, the position for the user binding and the users key information, creates a non-repudiable promise to include the data in the next log.

(10) Decentralized: CONIKS is not a decentralized system as the logs are not stored by very user. Instead it is a distributed system of CONIKS Servers hosted by a single service provider. The least number of CONIKS server needed per service provider are not specified but it is a tradeoff between the number of CONIKS server need to prevent a compromise and the cost of acquiring the severs.

(11) Avoids Proof of work: CONIKS avoids the proof of work clearly because some form of trust exists between the log servers and the user. To this note, there is no need for consensus on the content of the ledger. Also there are no coins minted in CONIKS hence no proof of work is needed to distribute money within the system. One more fact is that the logs are update within specified epoch by the log server. The epoch is determined by the server hence no need for proof of work to enforce timing for

update.

(12) Publicly Auditable: CONIKS implementation has auditors who audit the logs to ensure identity providers are not equivocating. They also ensure consistency through a gossip protocol to other auditors to ensure global consistency. Since the commitment hides the key data corresponding to a key data, the server in addition to the authentication path provides an opening of the commitment.

(13) Resilient to attack: If a compromised CONIKS Server tries to equivocate by allowing multiple binding for a single domain name, the server would in turn have to publish distinct logs to masquerade its tracks. But this would be detected during a gossip protocol between auditors. An attack to the system would involve compromising all the identity providers which would be relatively easier seeing that its a distributed system as compared to a decentralized system.

(14) Do not store entire log: CONIKS is efficient in bandwidth and storage, in that the user does not have to store the entire log as all they need to store is the Signed Tree root (STR) of the log. This STR is used for full verification as it is sufficient to check users key binding by ensuring the authentication path hashes given to the user sums up to the signed tree root the user stored and any changes or addition to the log can be proven.

A.3 Certificate Transparency

Certificate Transparency surfaced to address the insecurity in the issuance of SSL certificate. Notorious breaches such as that on Comodo Group, TrukTrust, ANSSI, DigiNotar and more have revealed how much damage mis-issued certificates can cause. The goal behind Certificate Transparency is to provide a public, verifiable, append-only log where trust is not required because the content can be verified cryptographically. It publicly auditable such that clients can check that certificates are in the log and servers can monitor the log for misused certificate. Certificate Transparency tries to remove the responsibility from the users as opposed to other solutions with a basic idea that the certificate is either in the log or not. If the certificate is not logged the users client would not proceed and if it is logged, the servers monitors to ensure the certificates are valid. Its soon to be wide adoption is majorly because the solution does not require a ground redesigning of the CA issuance ecosystem. There are three actors in the system- users, monitors and auditors. The users are the CA (responsible for entering the certificates into the log) and website servers. The monitors observe the log for invalid certificate or unauthorized certificate. Since log monitors fetch new entries added to the log they can act as backup read-only log. Auditor on the other hand verify the inclusion of a certificate in the log. They also proof that new entries were appended to the old Signed Tree Head (STH) by recalculating the hashes to ensure that the log server did not insert, delete or modify old entries. To ensure consistency among users, monitors and auditors, a gossip protocol is used between the auditor and monitor to exchange information about log.

(1) Append only: The log is a Merkel tree data structure where logs are chained to each other to obtain a single Merkel root, any change in previous record will result to a change in the Merkel root which would be detected.

(2) Non-Equivocate: A log server may attempt to present two different views of the log to auditors and monitors but this misbehavior would be detected by the auditors and monitors during the gossip protocol. But this security model works under the assumptions that the all auditors act honestly and do not collude with one another or the malicious log server.

(3) Transparency: The certificates are publicly visible and verifiable.

(4) Carbon dated messages: Due to the absence of proof of work in its core design,

information on the log cannot be said to be carbon dated to previous time.

(5) Avoids Proof of Work (PoW): The concept of Proof of Work exist due to the following reasons: the need to mint and distribute coins in the system, the need to reach a consensus on the order or content of the log, to prevent spam emails, to prevent double spend and the need to rate limit how often information is entered into the log to enhance the security of the log. In the case of CT these functions are not required and hence PoW is not incorporated into its design.

(6) Permission to Join: There are no permissions needed to join the network.

(7) User store entire log: Users in the system are required to store only the root of the tree known as the tree hash. Users can verify consistency of the tree among each other by simplify verifying that they have the same tree hash. Users can also perform full verification of the inclusion of a certificate by hashing the path form the leaf to the root.

(8) Accountability: The log server signs the root of the tree thereby being accountable for everything in the tree. This make it possible for the log server to be held responsible for any misbehavior.

(9) Liveness: In other for a client to be sure that an issued certificate by the Certificate authority was included in the log, a proof of inclusion should be attached to the new certificates. But this would make issuing of certificates slow as the log server would have to recalculate tree hash for each new certificate entry. CT solution to this is have the log server issue a Signed Certificate Timestamp (SCT) which is attached to the new certificate promising the client that the certificate would be included in the future log. A Maximum Merge Delay (MMD) is the maximum time in which the certificate must be included in the log.

(10) Decentralized: It is not a decentralized system but a distributed set of log servers who are not trusted.

(11) Resilient to attack: The more SCT are required in a certificate as each log issues SCT for a new certificate, the harder it would be for an attacker to compromise the system or go undetected since the attacker would not only have to compromise the certificate authority but also the log servers. They propose that each certificate should be included in at least two logs with five being the maximum for certificates that have lifetime of more than 39 months. The limited number of logs is to avoid increased time in issuing certificate and increased size of the TLS handshakes.

(12) Publicly Auditable: Client and auditors can request for proof of inclusion of a given certificates and monitors can check for misbehavior.

(13) Anonymity: Since certificates includes personal identifiable information, users are not anonymous in the system.

(14) Privacy Preserving: All certificates are logged in plaintext.

A.4 Naive Transparency Overlay

The basic security properties for any transparent system is consistency and non-frameability. Where consistency says a dishonest log server cannot potentially go undetected with presenting different views to the users within the system while non-frameability says dishonest users in the system cannot frame a log server for a misbehavior if it indeed behaved honestly. Participants in the network can therefore be reassured that they are seeing the same log and that they are interacting with the legitimate log server. According to the paper, Bitcoin Blockchain was said not to satisfy the basic security properties due to the fact that the miners who play a role synonymous to a log server (as they enter blocks into the blockchain) cannot be held responsible for any misbehavior. Therefore, their basic transparency overlay for Bitcoin has a log server in addition to the existing players (user and miners) in the system. The miners still mine block but those block are now sent to names set of trusted log servers who include contents into the log and produce signed evidence that transactions are stored in the log. Thus providing an extra layer that allows the blockchain to satisfy consistency and hold log server responsible for any misbehavior. The system has six actors: a sender, a receiver, a miner, a monitor, an auditor and a log server. The sender forms the transaction which is collated by the miner. The miner checks for any double spend transaction or invalid transaction and thereafter mine. The mined block is then sent to the log Server who then insert the block into the blockchain. The blocks should contain a set of transactions, a hash head of the block, the hash of the previous block and a height to impose the notion of timing.

(1) Non-Equivocate: A log server may attempt to present two different views of the log to auditors and monitors but this misbehavior would be detected by the auditors and monitors during the gossip protocol. But this security model works under the assumptions that the all auditors act honestly and do not collude with one another or the malicious log server.

(2) Append-only: Due to the fact that it follows the Merkel tree data structure where logs are chained to each other to obtain a single Merkel root, any change in previous record will result to a change in the Merkel root which would be detected.

(3) Transparency: Although the logs are maintained by distributed log servers, they are made public to all users with the system.

(4) Liveness: Provided the distributed named log server do not collude to perform a denial of service on a miner or block, transaction broadcast would eventually be added to the blockchain.

(5) Anonymity: There exist a kind of anonymity known as pseudo-anonymity where public keys are used to identity users within the system but this identity only exist within the bitcoin network and therefore are not tied to their real world identity but the distributed log servers are known named entity.

(6) Permission to join: The network is open to all.

(7) Avoids Proof of Work (PoW): Since it anchored on top of bitcoin, it is centered around PoW due to the following reason: one, since its not a centralized system, consensus has to be reached on the content of the block as well as the ordering of the block. Also the PoW acts as a form for minting and distributing coins in the system and for rate limiting the time for inserting a new block into the blockchain.

(8) Privacy Preserving: All transaction within the system are stored in plaintext.

(9) Carbon Dating: Due to the presence of proof of work in its core design, information on the log can be said to be carbon dated to previous time. Therefore, it would take a large amount of work to make changes to a message that was inserted days ago into the blockchain.

(10) Publicly Auditable: The log is publicly auditable by auditors and monitors in the system

(11) Accountability: The named log servers can be held accountable if they misbehave (modify or insert data in the middle of the blockchain). But this does not prevent the log server from dropping off messages. But provided the other set of log sever do not collude the block is bound to be accepted by one of the log server.

(12) Resilient to attack: In other to compromise the system the attackers need only to compromised the trusted set of named server. The greater the number of trusted server the harder it is for the attacker to compromised the system.

(13) Decentralized: it is termed a hybrid system as the mining of block is performed in a decentralized manner but the set of log servers are distributed named entities.

(14) Do not store entire log. In this system the users are required to store nothing except the user takes the role of a monitor as compared to storing the entire log.

A.5 Centrally Banked Cryptocurrency (RSCoin)

RSCoin is a cryptocurrency system that prevents double spend attacks and ensures the security of the system by relying on a distributed set of authorities while monetary supply and the constitution of the transaction ledger is maintained and controlled by the central bank. Despite the centralized monetary supply, RSCoin still provides strong transparency and auditability. This system addresses certain limitations with current cryptocurrency such as poor scalability and loss of control over monetary supply which could lead to extreme volatility in the value as currencies. The actors in this system are: The central bank, the mintettes and the users. The central bank is responsible for controlling the supply of money and maintaining the ledger. The mintettes on the other hand, are delegated institutions assigned by the central banks to validate transactions. This can be institutions with established relationship with the central banks such as commercial banks as they already have incentive to perform this task. The users can create pseudonyms identity and perform transaction within the network. The blocks controlled by the mintettes in which they store transaction of users within a certain specified epoch are known as Low-Level block. This Low-Level blocks are submitted to the central bank who then assembles them into one block known as the High-Level block. This High-Level block makes up the ledger for the system. The Low-Level block produced by each mintettes contains the transaction processed by that mintettes and the hash heads of all Low-Level blocks produced by other mintettes.

(1) Append-only: Due to the fact that it follows the Merkel tree data structure where logs are chained to each other to obtain a single Merkel root, any change in previous record will result to a change in the Merkel root which would be detected.

(2) Non-Equivocate: The system is designed to trust that the central authority would not equivocate on log information. To facilitate a more resilience system they propose adopting a broadcast system for distributing the log but this would come at a cost (high latency).

(3) Transparency: Both Low-Level and High Level blocks as well as the monetary supply are globally visible to all actors within the system.

(4) Liveness: Once a submitted transaction is validated by the mintettes, the mintettes issues a confirmation that the transaction would be included in the higher level block.

(5) Anonymity: There exist a kind of anonymity known as pseudo-anonymity where public keys are used to identity users within the system but this identity only exist within the network and therefore are not tied to their real world identity. Therefore, transactions are created, mined and validated anonymously.

(6) Permission to join: The network is open to all however; the users are assigned to various mintettes based on the hash of their transaction ID.

(7) Avoids Proof of Work (PoW): To avoid proof of work, consensus on valid transactions can be achieved with the use of named set of distributed mintettes assigned by the central bank. And it prevents the double spend attack by using a weak property that ensures that any transaction output must be featured in at most one transaction input. In this case, a basic consensus algorithm enforcing the ordering of the block used in current cryptocurrency to prevent double spend attack is not necessary. Also since minting and distribution of coins are performed by the central bank, the proof of

work is not needed. Rather than solving a cryptographic puzzle the mintettes simply collect transactions from users into a low level block.

(8) Privacy Preserving: Since RSCoin is a framework, it can support storing transactions in plaintext as well as adapting cryptographic techniques in existing systems such as Zero coin and Zero cash to ensure privacy for transactions.

(9) Carbon Dating: Due to the absence of proof of work in its core design, information on the log cannot be said to be carbon dated to previous time.

(10) Publicly Auditable: It has the notion of a timed personal audit where users can have access to the Lower-Level block within an epoch and audit it to observe if the behavior implied of the mintettes was the same when the user previously interacted with the mintettes. Also transactions in the Lower-Level block before it is submitted to the central bank can be audited by the users.

(11) Accountability: Since the set of mintettes are known entities they can be held responsible for any misbehavior. They also issue out receipt to users for a given submitted transaction which can be used to implicate the mintettes if they do not appear in the next block.

(12) Resilient to attack: The system can be attacked and compromised if the central bank is compromised by an attacker. Since no proof of work is required, the attacker would need only the signatures of

(13) Decentralized: It is somewhat distributed and centralized in that the monetary supply as earlier stated is controlled by the central bank while the validation of transactions is done by a distributed set of mintettes.

(14) Do not store entire log: The logs are kept and controlled by the central bank, therefore users of the system do not need to store the entire log but only block headers to

ensures that the log was not modified.

Most of the applications or solutions that use gossiping protocols to detect non-equivocation might not be an efficient solution as detecting misbehavior might be slow or might not happen at all if log server should collude with a majority of the auditors and monitors. An Efficient solution would be one that prevent equivocation such as that implemented by the Bitcoin Blockchain solution. This solution is efficient due to the following reasons. For one, since creating a fork in Bitcoin blockchain is very difficult, using this approach makes equivocation as hard as creating a fork. Also the use of Bitcoin Blockchain removes the need for trusted third party such as auditors and monitors.

Appendix B

Sample Code

This section contains some of the contracts deployed in this work. The exclusion of the Credit Default Swap contracts and the Collateral contracts (Ether and Token) from this section has nothing to do with it importance but to prevent this section from being too lengthy.

B.1 Credit Score contract

The Credit Score contract gives a overall view of all the credit event from all the contract linked to it. The contracts linked to it includes both the loan objects and the protection objects contracts. The functions within this contract are called by the contracts interfacing it.

```

pragma solidity ^0.4.12;

contract CreditScore
{

    /* A struct to hold the variables of each loan or bond */
```

```
  struct CreditHistory{
      address Cashprovider;
      address Cashtaker;
      uint amountDue;
      uint Instalments;
      uint currentmonth;
      uint [] coupons;
  }

  uint public numLoans;
  mapping (address => CreditHistory) public CreditHistories;
  event ActivatedEvent(address loanID, uint amount);
  event Length( uint lenght);


/* Function to create a new storage for a loan or bond contract*/
  function newloanHistory( address Cashprovider, address Cashtaker,
     uint amountDue, uint _instalments, address loanID) returns (
     address ){
      numLoans++;



      CreditHistories[loanID].Cashprovider=Cashprovider;
      CreditHistories[loanID].Cashtaker=Cashtaker;
        CreditHistories[loanID].amountDue=amountDue;
         CreditHistories[loanID].Instalments=_instalments;
          CreditHistories[loanID].currentmonth= 0;
           CreditHistories[loanID].coupons.push(0);

    return loanID;
  }

```

```
    /* Function to update the state of the variables during a monthly
       payment*/
    function populateHistory(address loanID, uint _amount)
    {


        CreditHistory storage CH= CreditHistories[loanID];
        CH.coupons.push(_amount);
        CH.currentmonth++;
        ActivatedEvent(loanID, _amount);
    }

  /* A getter function to return the state of the loan */
    function displayHistory(address loanID) public constant returns (
       address, address, uint, uint, uint, uint[]){
        return(CreditHistories[loanID].Cashprovider, CreditHistories[
           loanID].Cashtaker,
        CreditHistories[loanID].amountDue,CreditHistories[loanID].
           Instalments,
        CreditHistories[loanID].currentmonth,CreditHistories[loanID].
           coupons);

    }

  /* A function to return an amount given a specific month */
     function getAmount( address investor, address loanID, address owner
        ,  uint monthno) returns (uint Amount){
        if (CreditHistories[loanID].Cashprovider==investor &&
           CreditHistories[loanID].Cashtaker==owner){
             Amount= CreditHistories[loanID].coupons[monthno];
        }
        return Amount;
    }


  /* A function to obtain the length of an array given the loan address
      */
    function getlength(address loanID) returns (uint)
    {
        uint lent = CreditHistories[loanID].coupons.length;
        return lent;
        Length(lent);
    }

    /*A function to change ownership */
    function changename(address cdsseller, address loanID){

        CreditHistories[loanID].Cashprovider = cdsseller;


    }
}
```

B.2 Loan contract

This contract describes a basic loan that allows a lender to lend money to a borrower. Buried within it are functions that allow the debtor to pay the lender on a monthly basis and also interface with the Credit score contract to update it with the values for each month that denotes either a default occurred or not. It imports the an oracle called Oraclized that feeds the price of ETH in USD into it.

```
pragma solidity ^0.4.8;

import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";
```

```

contract CreditScore {


    function newloanHistory( address Cashprovider, address Cashtaker,
       uint amountDue, uint _instalments, address loanID) returns (
       address );

    function populateHistory(address loanID, uint _amount);

    function getlength(address loanID) returns (uint);

}

contract Loan is usingOraclize {

 /* Variables for the Loan contract */

    address public Cashprovider;
    address public Cashtaker;
    uint256 public Principal;
    uint256 public amountLeft;
    uint public interest;       // modified the interest tot a fixed
       amount
    uint public Instalments;            // how much to be paid per
       instalments
    uint public startDate;
    uint public endDate;
    address public loanID;
    mapping (address => uint) public Balance;
    enum State {Initialized, Release}
    State public status;
    address myAddress;
    bool public active;
    bool public Defaulted;
    bool check;
    uint public amountDue ; //total amount

    address _CreditScore = 0x997480e4F88c69f8CAd1E6AE0aEbA2e6f88Cdd01 ;
       // initialize this before running the code

    CreditScore userHisory = CreditScore(_CreditScore);

    event DeFaulted (bool defaulted);


     modifier onlyCashprovider()
    {
        if (msg.sender != Cashprovider) throw;
        _;
    }

    modifier onlyCashtaker()
    {
        if (msg.sender != Cashtaker) throw;
        _;
    }



    /* code to act as a calender for the monthly payment */

    uint [13] public array;

    uint public counter;

    function setCalender(){

    array[0]= now ;
    array[1]= now  + 3 minutes;
    array[2]= now  + 6 minutes;
    array[3]= now  + 9 minutes;
    array[4]= now  + 12 minutes;
    array[5]= now  + 15 minutes;
    array[6]= now  + 18 minutes;
    array[7]= now  + 21 minutes;
    array[8]= now  + 24 minutes;
    array[9]= now  + 27 minutes;
    array[10]= now  + 30 minutes;
    array[11]= now  + 33 minutes;
    array[12]= now  + 36 minutes;
    }


   /* Oraclize begins here */
    uint public EthtoUSD;

    event newOraclizeQuery(string description);
    event newEtherPrice(string price);

    function EthPrice() {
        update(); // first check at contract creation
    }

    function __callback(bytes32 myid, string result) {
        if (msg.sender != oraclize_cbAddress()) throw;
        newEtherPrice(result);
        EthtoUSD = parseInt(result, 2); // let's save it as $ cents

    }

    function update() payable {
        newOraclizeQuery("Oraclize query was sent, standing by for the
            answer..");
        oraclize_query("URL", "json(https://min-api.cryptocompare.com/
            data/price?fsym=ETH&tsyms=BTC,USD,EUR).USD");
    }

/* constructor function */

  function Loan (address _borrower, uint _rate, uint _blocknum, uint256
      _amount, uint _instalments)
  {
    myAddress = this;
    Cashprovider = msg.sender;
    Cashtaker = _borrower;
    Principal = _amount;
    interest = _rate;
    endDate = _blocknum;
    Instalments= _instalments;
    status = State.Initialized;
    amountDue= Principal + interest;
    EthPrice();
  }

  /* Cashprovider sends money to the contract */
  function ()onlyCashprovider payable{

  }
```

```

/* The function is triggered by the debtor in other to withdraw money
    from the contract */
function receiveMoney() onlyCashtaker payable returns (bool res) {
  if (status != State.Initialized) {return;}

  if (myAddress.balance >= Principal)
  {
          status = State.Release;
          active= true;
          startDate = now;
          setCalender();
          if(!Cashtaker.send(Principal)) throw;
    loanID = userHisory.newloanHistory(Cashprovider, Cashtaker,
        amountDue,  Instalments, myAddress );
    return true;
      }
      else throw;


}

// if the contract was not utilized by the debtor after certain number
     of blocks, kill the contract
// in the future this function would implement a modifier function
function Kill() onlyCashprovider
{
      if (status == State.Release) throw;

      suicide (Cashprovider);
}

```

```
/* Function to pay the cash provider monthly */
function payback() onlyCashtaker payable returns (bool res)
{


  if (endDate < block.number) throw;

  if (msg.value < Instalments && msg.value > 0)
  {


    amountLeft = Instalments - msg.value;
    reportDefault (amountLeft);
    check= true;
    if (!Cashprovider.send(msg.value)) throw;

  }

  else if (msg.value == Instalments)
  {
    check=true;
    reportDefault (0);
    if (!Cashprovider.send(msg.value)) throw;

  }

  else throw;

}

 /* function to report if a default by the organization and update the
     CreditScore object */
```

```
function reportDefault (uint _amount)
{

      uint i;
      uint j;

      if(msg.sender!=Cashtaker)
  {

          counter=0;
    for (i = 0 ; i <= 12 ; i++)
    {

      if (now > array[i])
      counter ++;

          }
      }
      uint lent = userHisory.getlength(loanID);
      if (lent == counter)
      {

      }

      else if (lent < counter)
      {
          for (j=lent ; j<counter ; j++)
    {
              userHisory.populateHistory(loanID, _amount);
          }

      }
```

```
}

else
{
    counter=0;
     for (i=0 ; i<=12 ; i++)

      {
          if (now > array[i]){
          counter ++;

          }
      }

      lent = userHisory.getlength(loanID);
      if (lent == counter)
          {
              userHisory.populateHistory(loanID, _amount);
          }

      else if (lent < counter)
      {
          for (j=lent ; j<counter ; j++)
    {
              userHisory.populateHistory(loanID, Instalments);
          }
              userHisory.populateHistory(loanID, _amount);
```

```
        }




    }

}



}
```

B.3 Bond contract

A bond is simply a debt obligation where investors who buy corporate bonds are lending money to the company that issued the bond. This contracts contains functions that model the flow of a typical everyday bond with the exception that its written in solidity language as opposed to natural language. This contracts interfaces with the Credit score contract by updating it with values each month denoting either a default occurred or not. It imports the an oracle called Oraclized that feeds the price of ETH in USD. In addition it contains the standard token deployed by the Ethereum community.

```
pragma solidity ^0.4.7;

import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract CreditScore {


    function newloanHistory( address Cashprovider, address Cashtaker,
        uint amountDue, uint _instalments, address loanID) returns (
        address );

    function populateHistory(address loanID, uint _amount);

    function getlength(address loanID) returns (uint);

}

contract tokenRecipient { function receiveApproval(address _from,
    uint256 _value, address _token, bytes _extraData); }

contract MyToken {
    /* Public variables of the token */
    string public standard = 'Token 0.1';
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;

    /* This creates an array with all balances */
    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public allowance;

    /* This generates a public event on the blockchain that will notify
        clients */
    event Transfer(address indexed from, address indexed to, uint256
        value);

    /* Initializes contract with initial supply tokens to the creator of
         the contract */
    function MyToken(
        uint256 initialSupply,
        string tokenName,
        uint8 decimalUnits,
        string tokenSymbol
        ) {
        balanceOf[msg.sender] = initialSupply;              // Give the creator all initial tokens
        totalSupply = initialSupply;                        // Update total supply
        name = tokenName;                                   // Set the name for display purposes
        symbol = tokenSymbol;                               // Set the symbol for display purposes
        decimals = decimalUnits;                            // Amount of decimals for display purposes
    }

    /* Send coins */
    function transfer(address _to, uint256 _value) {
        if (balanceOf[msg.sender] < _value) throw;           // Check if the sender has enough
        if (balanceOf[_to] + _value < balanceOf[_to]) throw; // Check for overflows
        balanceOf[msg.sender] -= _value;                     // Subtract from the sender
        balanceOf[_to] += _value;                            // Add the same to the recipient
        Transfer(msg.sender, _to, _value);                   // Notify anyone listening that this transfer took place
    }

```

```
     function balanceToken(address _user) constant returns (uint256
        balance)
    {
        return balanceOf[_user];
    }

    /* Allow another contract to spend some tokens in your behalf */
    function approve(address _spender, uint256 _value)
        returns (bool success) {
        allowance[msg.sender][_spender] = _value;
        return true;
    }

    /* Approve and then comunicate the approved contract in a single tx
       */
    function approveAndCall(address _spender, uint256 _value, bytes
       _extraData)
        returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, this, _extraData
                );
            return true;
        }
    }

    /* A contract attempts to get the coins */
    function transferFrom(address _from, address _to, uint256 _value)
       returns (bool success) {
        if (balanceOf[_from] < _value) throw;                  // Check
            if the sender has enough
        if (balanceOf[_to] + _value < balanceOf[_to]) throw;  // Check
            for overflows
        if (_value > allowance[_from][msg.sender]) throw;   // Check
            allowance
        balanceOf[_from] -= _value;                          // Subtract
             from the sender
        balanceOf[_to] += _value;                            // Add the
            same to the recipient
        allowance[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    }

    /* This unnamed function is called whenever someone tries to send
        ether to it */
    function () {
        throw;     // Prevents accidental sending of ether
    }
}



contract comBond is usingOraclize {

    /* Varaibles for the Bond contracts */

    uint256 public couponRate = 0;  //interest rate in ether
    uint public startDate= 0;
    uint public Maturity= 0;
    uint public tokenPrice;
    uint public amountnow;
    mapping (address => uint256) balanceOf;  // each address in this
        contract may have tokens.
    mapping (address => uint256) balances;
    address  Cashtaker;                // the owner is the creator of
        the smart contract
    MyToken public Token;
    address myAddress;
    uint payment;

    address [] listofloan;
    string public defaulted = "No";

    address _CreditScore = 0xa ; // initialize this before running the
        code (holds the address of the credit score contract)

    CreditScore userHisory = CreditScore(_CreditScore);

    modifier onlyCashtaker()
    {
        if (msg.sender != Cashtaker) throw;
        _;
    }

    /* This generates a public event on the blockchain that will notify
        clients */

    event BuyXYZ(string note, address indexed recipient, uint256 value);
    event Defaults(string note, address indexed recipient );
    event PartialDefault(string note, address indexed recipient , uint
        amount);

    // code to act as a calender for the monthly payment

    uint [13] public array;

    uint public counter;

    function setCalender(){

    array[0]= now ;
    array[1]= now  + 3 minutes; // this was used in minutes for testing
       but can be easily replaced with days
    array[2]= now  + 6 minutes;
    array[3]= now  + 9 minutes;
    array[4]= now  + 12 minutes;
    array[5]= now  + 15 minutes;
    array[6]= now  + 18 minutes;
    array[7]= now  + 21 minutes;
    array[8]= now  + 24 minutes;
    array[9]= now  + 27 minutes;
    array[10]= now  + 30 minutes;
    array[11]= now  + 33 minutes;
    array[12]= now  + 36 minutes;
    }


    /* Oraclize begins here */
    uint public EthtoUSD;

    event newOraclizeQuery(string description);
    event newEtherPrice(string price);

    function EtherPrice() {
        update(); // first check at contract creation
    }

    function __callback(bytes32 myid, string result) {
        if (msg.sender != oraclize_cbAddress()) throw;
        newEtherPrice(result);
        EthtoUSD = parseInt(result, 2); // let's save it as $ cents
        // do something with the USD price
    }

    function update() payable {
        newOraclizeQuery("Oraclize query was sent, standing by for the
            answer..");
        oraclize_query("URL", "json(https://min-api.cryptocompare.com/
            data/price?fsym=ETH&tsyms=BTC,USD,EUR).USD");
    }

    /* constructor function */

    function comBond(uint256 _tokenPrice, uint256 _couponRate, uint
       _startDate, uint _maturity, MyToken _mytokenaddress){

        couponRate= _couponRate * 1 ether;
        startDate= _startDate;
        Maturity= _maturity;
        tokenPrice= _tokenPrice * 1 ether;
        Token= MyToken(_mytokenaddress);
        myAddress = this;
        Cashtaker=msg.sender;
    }

   /* function that enables investor buy bonds from the organization */

```

```
    function Purchasebond() payable
    {
        if (msg.sender != Cashtaker){

        if (block.number < startDate || block.number > Maturity) throw;

        uint256 amount= msg.value;
        balances[msg.sender]= amount;
        amountnow += amount;
        uint256 value = amount / tokenPrice;
        balanceOf[msg.sender] = value;
        Token.transfer(msg.sender, value);
        address loanID;
        loanID = userHisory.newloanHistory(Cashtaker, msg.sender,
            amountnow, 0, myAddress);
        listofloan.push(loanID);
        BuyXYZ("A token has been purchased", msg.sender, msg.value);
            // if the sender is the fund manager then that would mean
            funds are requires to pay out for the sale of tokens and
            therefor this ether does not need to be allocated to any
            specific user, it will be paid out
        }

    }


    /* withdraws all ether from contract that is then used to purchase
        assets according to the current portfolio */
    function withdrawETH() onlyCashtaker returns (bool success)
    {
        setCalender();
        if(!Cashtaker.send(this.balance)) throw;
        return true;
    }

/* pay interest to the investors on a monthly basis */
    function Repay() payable returns (bool success)
    {
        if (Maturity < block.number) throw;
        uint256 bal = Token.balanceToken(msg.sender);
        uint256 zeroCoupon = bal * couponRate;
        uint256 amount = bal * tokenPrice;
        payment = (zeroCoupon + amount)/12;



         if (myAddress.balance >= payment){
        balanceOf[msg.sender] == 0;
        reportDefault (0, msg.sender);
        if (!msg.sender.send(payment)) throw;
        return true;
         }

         if (myAddress.balance < payment){
            balanceOf[msg.sender] == 0; // this is set to zero so that
                the user wont run the function again
            uint duenow= payment - myAddress.balance;
            reportDefault (duenow, msg.sender);
            if (!msg.sender.send(payment)) throw;

            defaulted="Partial Default";
            PartialDefault("The organization has defaulted on
                partpayment to this address", msg.sender, duenow);
         }

         if (myAddress.balance == 0 ){
             if ( balanceOf[msg.sender] > 0)
                {
            reportDefault (payment, msg.sender);
           defaulted="Full Default";
           Defaults("The organization has defaulted on full payment to
               this address", msg.sender);
                }

         }


    }

    /* function to report if a default by the organization an update the
         CreditScore object */
  function reportDefault (uint _amount, address loanID)
  {

        uint i;
        uint j;

        if(msg.sender!=Cashtaker){
            counter=0;
        for (i = 0 ; i <= 12 ; i++)
        {

            if (now > array[i]){
            counter ++;

            }
        }
        uint lent = userHisory.getlength(loanID);
        if (lent == counter)
            {

            }

        else if (lent < counter)
            {
                for (j=lent ; j<counter ; j++){
                userHisory.populateHistory(loanID, _amount);
                }

            }





  }

  else {
      counter=0;
       for (i=0 ; i<=12 ; i++)

        {
            if (now > array[i]){
            counter ++;

            }
        }

        lent = userHisory.getlength(loanID);
        if (lent == counter)
            {
                userHisory.populateHistory(loanID, _amount);
            }

        else if (lent < counter)
            {
                for (j=lent ; j<counter ; j++){
                userHisory.populateHistory(loanID, payment);
                }
                userHisory.populateHistory(loanID, _amount);
            }




    }
  }

}
```

Bibliography

[1] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

[2] A. Back et al. Hashcash-a denial of service counter-measure, 2002.

[3] P. Bailis, A. Narayanan, A. Miller, and S. Han. Research for practice: cryptocurrencies, blockchains, and smart contracts; hardware for deep learning. *Communications of the ACM*, 60(5):48–51, 2017.

[4] J. Benaloh. M. de mare efficient broadcast time-stamping technical report 1. *Clarkson University Department of Mathematics and Computer Science*, 1991.

[5] J. Benaloh and D. Tuinstra. Receipt-free secret-ballot elections. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 544–553. ACM, 1994.

[6] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Anonymity for bitcoin with accountable mixes. *Preprint*, 2014.

[7] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform. Technical report, Ethereum, 2014.

[8] D. Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[9] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. Ryan, E. Shen, and A. T. Sherman. Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. *EVT*, 8:1–13, 2008.

[10] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[11] K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[12] J. Clark and A. Essex. Commitcoin: Carbon dating commitments with bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 390–398. Springer, 2012.

[13] CodeTract. Inside an ethereum transaction, 2015.

[14] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. *Transactions on Emerging Telecommunications Technologies*, 8(5):481–490, 1997.

[15] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[16] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[17] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[18] S. Eskandari, J. Clark, D. Barrera, and E. Stobert. A first look at the usability of bitcoin key management. *Workshop on Usable Security (USEC*, 2015.

[19] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International Conference on Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[20] A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. In *Advances in CryptologyAUSCRYPT'92*, pages 244–251. Springer, 1993.

[21] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[22] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[23] R. Haenni and O. Spycher. Secure internet voting on limited devices with anonymized dsa public keys. *EVT/WOTE*, 11, 2011.

[24] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[25] B. Joseph, M. Andrew, J. Clark, N. Arvind, K. Joshua, and F. Edward. Research perspectives and challenges for bitcoin and cryptocurrencies. *IEEE Symposium on Security and Privacy*, 2015.

[26] A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant electronic elections. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, pages 61–70. ACM, 2005.

[27] A. Juels, A. Kosba, and E. Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 283–295. ACM, 2016.

[28] A. Kiayias and M. Yung. Self-tallying elections and perfect ballot secrecy. In *International Workshop on Public Key Cryptography*, pages 141–158. Springer, 2002.

[29] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[30] P. McCorry, S. F. Shahandashti, and F. Hao. A smart contract for boardroom voting with maximum voter privacy. *Financial Cryptography*, 2017.

[31] R. C. Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[32] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. *URL: http://www. bitcoin. org/bitcoin. pdf*, 2008.

[33] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[34] R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. *IACR Cryptology ePrint Archive*, 2016:454, 2016.

[35] T. Ruffing, P. Moreno-Sanchez, and A. Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *European Symposium on Research in Computer Security*, pages 345–364. Springer, 2014.

[36] P. Y. Ryan, D. Bismark, J. Heather, S. Schneider, and Z. Xia. Prêt à voter: a voter-verifiable voting system. *IEEE transactions on information forensics and security*, 4(4):662–673, 2009.

[37] K. Sako and J. Kilian. Receipt-free mix-type voting scheme. In *Advances in CryptologyEUROCRYPT95*, pages 393–403. Springer, 1995.

[38] D. Sandler and D. S. Wallach. Casting votes in the auditorium. *EVT*, 7:4–4, 2007.

[39] Springer. *A Survey of Attacks on Ethereum Smart Contracts (SoK)*, 2017.

[40] N. Szabo. The idea of smart contracts, 1997.

[41] S. Thompson, P. Lamela Seijas, and D. Adams. Scripting smart contracts for distributed ledger technology. *Cryptology ePrint Archive*, 2016.

[42] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Project Yellow Paper, 2014.

[43] F. Zagórski, R. T. Carback, D. Chaum, J. Clark, A. Essex, and P. L. Vora. Remotegrity: Design and use of an end-to-end verifiable remote voting system. In *International Conference on Applied Cryptography and Network Security*, pages 441–457. Springer, 2013.