# Toward Cryptocurrency Lending

Mildred Chidinma Okoye[1,2] and Jeremy Clark[1(✉)]

[1] Concordia University, Montreal, Canada
j.clark@concordia.ca
[2] Deloitte, London, UK

**Abstract.** Lending has been posited as an application of blockchain technology but it has seen little real deployment. In this paper, we discuss the roadblocks preventing the effortless lending of cryptocurrencies, and we survey a number of possible paths forward. We then provide a novel system, Ụgwọ, consisting of experimental smart contracts written in Solidity and deployed on Ethereum to demonstrate how a decentralized lending infrastructure might be constructed.

## 1 Introductory Remarks

Lending has been posited as an application of blockchain technology but we have seen little real deployment of lending. In Sect. 2, we discuss roadblocks and possible paths forward. We do this in service of other researchers who might want to look at this issue—we view our own contributions as an initial look and not the final word in this complex area. We outline our agenda in a few steps: (1) we review the role of lending in a modern economy, (2) we identify the key tensions between cryptocurrencies like Bitcoin and Ethereum and lending, (3) we review proposals for lending, and (4) we suggest how to move forward. In Sect. 3, we present our lending infrastructure Ụgwọ which incorporates the points we discuss. Ụgwọ is designed to be flexible and extensible; traditional fiat-based lending is not one-size-fits-all and consists of a patchwork of loan structures, instruments, and intermediaries. We show some basic types of loans and basic types of risk mitigation as examples of what could be added to Ụgwọ to support an infrastructure for lending.

## 2 A Research Agenda for Cryptocurrency Lending

### 2.1 The Role of Lending in a Modern Economy

It is difficult to overstate the role of lending in a modern economy. Take, as an illustrative example, the role of a central bank; one of the main national institutes (along with the treasury) that cryptocurrencies aim to displace. First and foremost, a central bank is an actual bank, providing accounts for its member banks to deposit money and earn interest. Member banks provide interest-earning accounts to the public. Interest is paid to the public because banks use


A. Zohar et al. (Eds.): FC 2018 Workshops, LNCS 10958, pp. 367–380, 2019.
https://doi.org/10.1007/978-3-662-58820-8_25

the deposited money to form loans. Because central bank interest rates are low, banks prefer to lend to other banks any excess cash they hold at day's end instead of depositing them (other banks borrow to meet liquidity requirements). These loans earn interest, and central banks target this specific lending rate when they intervene in the economy. The most common intervention is the buying (circulating new money) or selling (removing circulating money) of government bonds, which are interest-earning loans from investors to the government. Central banks will also provide loans (of 'last resort') to banks unable to secure loans from other banks, typically during some sort of liquidity crisis. An economy without loans would have no interest rates, no bonds, and essentially nothing for a modern central bank to do.

### 2.2 Two Critical Issues for Lending with Cryptocurrencies

The crypto-economy is effectively an economy without loans. We identify two primary roadblocks:

- **Monetary instability.** While a loan might be in anything of value, it is typically done with money. Cash loans work best when the value of the money is relatively stable. By contrast, cryptocurrencies have historically appreciated in value over time (as of the time of writing). In a lending situation, this means the cash taker will end up owing far more than he borrowed. If the scenario were reversed and the currency depreciated rapidly, the cash provider would prefer to spend the money rather than locking it up in a loan where it will shed value over time. Even without long-term upward or downward drifts in value, short-term volatility adds risk to a loan for both the cash taker and the cash provider.
- **Counter-party risk.** While the hype surrounding blockchain technology centers on how it can enable trustless financial systems, there is no way to blockchain your way out of counter-party risk. If Alice truly lends money to Bob—truly in the sense that Bob fully owns it and can do with it as he pleases—then Bob can abscond with the money.

### 2.3 Existing Proposals

A number of companies have launched loan products or systems based on cryptocurrencies. In the most common architecture, a central company arranges loans and the loans are simply denominated in cryptocurrencies like Bitcoin. These services vary from at interest bearing accounts to peer-to-peer lending for investment purposes to social justice orientations like mirco-lending for the unbanked or the subprime market. As opposed to our system Ụgwọ, these do use smart contracts to structure the actual loans.

### 2.4 Dealing with Monetary Instability

We summarize a few suggestions for adding stability to cryptocurrencies.

- The rate of release of new currency into the system could be modified to enable new currency to be introduced at (i) a more insightful rate or (ii) based on some internal metrics of the system like number of transactions. [*Remark:* an insightful rate has been elusive despite many alt-coins customizing the schedule and it is difficult to see how metrics could not be gamed].
- A cryptocurrency can also use explicit pegging but it is no better suited to this system than standard currencies.
- A central bank could manage currency circulation while allowing other aspects to be decentralized [4]. [*Remark:* Central banks have been historically unsuccessful at using money circulation as a target [7]].
- The loan could be use the cryptocurrency as the medium of exchange but use a stable (*e.g.,* government) currency as the unit of account.

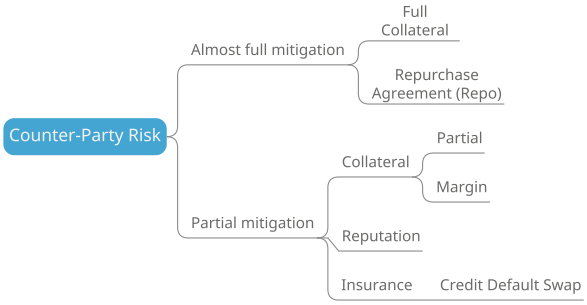


**Fig. 1.** Standard approaches to dealing with counter-party risk.

In Ụgwọ, we use the last approach. In other words, a loan could be $100 USD paid in Ether at the exchange rate at loan time and repaid 3 months later at $110 USD paid in Ether at the new exchange rate. This approach requires the smart contract to be aware of the exchange rate which introduces a trusted third party, called an oracle [11] and is discussed further in the next section.

### 2.5 Dealing with Counter-Party Risk

In Fig. 1, we outline the basic approaches from finance for dealing with counter-party risk.

- *Full Collateral:* It is common for Bitcoin-based solutions, *e.g.,* for fair exchange [1,3,10] or payment channels [5,9], to deal with counter-party risk by requiring full collateral. This is a simple approach but one unlikely to scale to an entire economy: economic actors are chagrinned to leave money where it earns no interest and economic benefit.

- *Repurchase Agreement:* A loan collateralized fully with same currency as the loan is not a loan therefore collateral only works if it is something different of the same value. If this something is on-blockchain (say a token representing something of value), the cash provider can have the collateral sit locked up in escrow (where it benefits neither the cash provider or taker) or could take full ownership of the collateral with the promise of returning it when the loan is repaid. This is a repurchase agreement and is common when the cash provider is perceived to be at less risk of absconding than the cash taker.
- *Partial Collateral:* The cash taker might stake something of lesser value than the loan in collateral, a third party to a loan might use partial collateral to insure a loan (see below), or sometimes loans are internal to a system such as leveraged positions in financial markets where the manager can liquidate the loan if the partial collateral (margin) dissipates due to market conditions.
- *Reputation:* A more abstract form of collateral is one's reputation and lending history. The difficulty with reputation is that it requires strong identities, something missing from decentralized currencies, as rogue entities can regenerate a new identity if the reputation of their old identity suffers and they can generate fake histories by lending to themselves with fake identities. These are not impossible to address but are difficulties.
- *Insurance:* Consider the case where Alice lends to Bob and does not trust him. If Alice trusts Carol and Carol trusts Bob, then Carol could insure the loan. Of course, Carol in this case could also just lend the money to Bob but there are a few scenarios where she might let Alice lend the money. One is if Carol's assets are not liquid. A second is that Carol might employ partial collateral: she could insure 100 loans of similar value but only stake 10% of the lent money as a margin against defaults. This costs her less than making the loans herself, and provides the cash providers insurance assuming the default rate is less than 10%. One standard financial instrument to implement this type of insurance, with some additional complexities discussed later, is a credit default swap (CDS).

## 3 The Ụgwọ Lending Infrastructure

Ụgwọ is an extensible system of smart contracts to enable different types of lending on Ethereum.[1] It is centred around recording credit events—when a party fails to fulfill the terms written in a loan contract—in a common ledger called a Credit Event object. We considered two implementation approaches:

- *Internal Variable.* In one approach, a loan has a credit event object within itself where the credit event is a variable contained within the loan contract. The issue with this approach is one of encapsulation: any external contract protecting the loan (via insurance or collateral) would have to reach inside the loan object when all it needs to know to function is whether a credit event occurred or not.

[1] https://github.com/MildredOkoye/Ugwo.

- *Object Oriented Approach.* It would be interesting if the Credit Event object sat at its own address such that protection contracts could be externally deployed and would not have to be worried about each loan that they insure individually. Protection would be external contracts and would just have a global view of all credit events from a single address given specific loan identifier (such as the loan address). To ensure compatibility, we can use interfaces which in object oriented programming specify the functions that must exist. Interfaces are similar to abstract classes in that they do not have any definition of functions contained within. An interface provides developers a guide as to how to implement the contract. Thus the Credit Event object is the core of extendable system where new loan types can be added and new protection types.

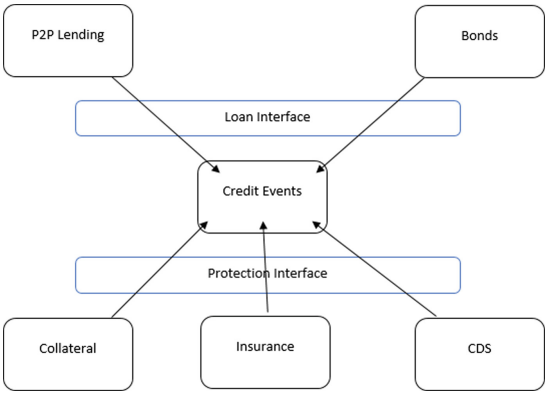


**Fig. 2.** The Ụgwọ lending infrastructure showing how the various loan and protection objects interface with the Credit Event object.

In Ụgwọ, we implement 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. These interfaces and their links to the loan objects are shown in Fig. 2.

### 3.1 Overview of Loan Objects

**Peer to Peer Lending.** We start with a basic loan contract constructed by the cash provider. The loan has parameters such as the address of the cash provider and cash taker, the principal amount to be lent, the start and end dates of the loan, the repay value and repay schedule. The cash provider runs the constructor and funds the contract. The cash taker runs a function in the loan contract to retrieve the principal in the contract. At maturity, the borrower calls a function

to pay back the principal with the corresponding interest. As with all of our objects, modifiers ensure that only the stated party can run each a function in a contract, and an internal state machine governs at which phase of the contract each function can be called. 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 does not retrieved the money after a timeout.

If the cash taker fails to pay back the loan within the timeframe, the loan object itself cannot transition states without someone calling a function. In Ụgwọ, a 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 the Credit Event object discussed below. This is how a loan moves into a default state and it relies one someone having an incentive to transition the loan (otherwise it is likely inconsequential if it sits dormant).

**Bonds and Commercial Paper.** We implement a simple 'zero coupon' bond. The contract uses an external library implementing EIP20 tokens.[2] The cash taker, generally an organization or corporation in this case, creates a set of tokens that represent units of cash it will accept (and later repay) from individual cash providers. The cash taker runs the constructor (with variables for start date, end date, bond value, repay value, *etc.*) and funds the contract with tokens. A function is used to accept payment from investors where tokens representing the amount borrowed 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 all bonds sold. The bond is a bearer bond in the sense that the bond contract does not track the addresses of who owns each bond. 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 submitted 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 the organization defaults on its payment after the due term.

### 3.2 Overview of Protection Objects

**Collateral.** Two types of collateral are defined in Ụgwọ—a token collateral and an ether collateral. A token collateral contract accepts a EIP20 token which might represent a token from a ICO, DAO-style contract, loan contract or anything else with value that the cash provider is willing to accept. The constructor function of the contract states the amount of tokens the cash taker is willing to put up as collateral. A separate function allows the cash taker to instantiate all agreements with the cash provider; they were not included in the constructor

[2] https://github.com/ConsenSys/Tokens.

function to allow the collateral function to be run by any investor. If at the end of the term the cash taker defaults, a function to get the token out of escrow can be run by the cash provider. The function first checks for a credit event, or triggers a credit event if the conditions for a default are met. An ether collateral accepts ether as collateral—since the loan itself is in Ether, this is useful for partial collateral functions or when the collateral is backing insurance rather than a loan.

**Credit Default Swaps.** A credit default swap (CDS) is an agreement between two parties (a seller and a buyer) where the CDS seller fulfils the debt of a loan to the CDS buyer if a credit event occurs on the loan. The CDS seller then takes ownership of the loan. If more there is more than one seller of a CDS per loan (as is permitted and common in financial markets for speculation), the loan is auctioned and the market clearing price is used to settle the swaps.[3] A CDS seller subsumes the same risk position as the actual cash provider in the loan but the benefit to the CDS seller is not having to liquidate any assets (she can have effectively no cash on hand if an event never happens). The benefit to the cash provider is that a loan with a CDS only defaults if both the cash taker and the CDS seller default.

CDSs have a bad reputation after the 2008 financial crisis in the United States, where the CDS market was unlit and considered by many to be under-regulated. In Ụgwọ, the CDS market is transparent and CDS buyers can have enforced reserves that automatically settle with CDS buyers when a credit event occurs on an insured loan. CDS sellers themselves an be given a Credit Event object. Our implementation is rudimentary (without naked CDSes, auctions, or other features) and we expect that a full-fledged, decentralized CDS market would constitute an entire research paper by itself.

### 3.3 Overview of the CreditEvent Object

It would be simpler to implement a CreditEvent object within each loan (P2P or Bond) contract. One reason 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 monitored.

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

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

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 Ụgwọ, each loan's constructor 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 cash provider and a default written to the CreditEvent contract. A value of zero is written if the amount being paid covers the total amount or is in excess (in this case, the surplus is returned to the cash taker). 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.

## 4 Discussion

### 4.1 Exploring 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 such as a URL, 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 on Ethereum this way. This is because the blockchain follows a consensus-based model that ensures all inputs can be validated. 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.

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 side-step the monetary instability of Ether which makes it unattractive for lending. Thus in nominal terms, the amount of ether being paid back might be more or less than the amount borrowed depending on whether it's value increased or decreased relative to the fiat 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 service has emerged, called an oracle, which is trusted external entity that puts data onto the blockchain where it can be accessed by other contracts. In Ụgwọ, we use Oraclize[4] to feed the exchange rate of Ether with USD into our 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 order to feed the current exchange value into the blockchain, a link to any exchange has to be manually inserted into the oracle's code; 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. Finally oracles are trusted parties that can lie about the exchange rate and collude with cash takers to steal from cash providers. We remark that oracles do have a reputation and in most countries, stealing is still subject to legal recourse even if it is on a blockchain.

### 4.2 Automatic Actions

Many Ethereum beginners have to adjust their mental model of smart contracts to the fact that a contract will not run unless if one of its functions is called. It cannot automatically perform actions, say, after some period of time has passed. In Ụgwọ, loans like bonds have a default function that checks if there has been a default by the cash taker. This default function has to be triggered by someone in order to default the loan and update the CreditEvent object. An option is to use the Ethereum Alarm Clock[5] to trigger the function monthly. It is a trusted third party 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. The drawback is its heavy integration with the loan contract, as well as arranging payments to the service. 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 cash provider. Due to the fact that the insurance or collateral can only be claimed after a default occurs, the cash provider in the contract would have more incentive to run the function every month. Hence, we did not deploy the alarm clock.

### 4.3 Implementing the Monthly Array Object

To implement a monthly payment, we could reference either time (*e.g.,* `now` or `block.timestamp`) or block interval (*e.g.,* `block.number`). Timestamps are not reliable and be manipulated by miners. This is due to the decentralization of the system; there is no wall clock for reference and node's local clocks can never be

[4] https://github.com/oraclize.
[5] http://www.ethereum-alarm-clock.com/.

perfectly synchronized (*i.e.,* to the millisecond). Ethereum permits a 900 ms lead or lag in time. When using block numbers, there is also a lack of precision. One could estimate that a 31 day month would be something like 179 759 blocks.[6] While this is a challenge for applications that need near real-time fidelity but for loan payments, we would argue that time slippage is not critical for loans. We utilize time not blocks. If a loan lies dormant for longer than a month, with Ethereum's model of function-initiated state changes, the loan's state will not change. However the next function to be called, whether a payment or default check, will update the previously skipped months in CreditEvent while writing the current result of the called function.

### 4.4 Implementing the CreditEvent Contract

Choosing an appropriate data structure for CreditEvent presented some challenges. We want loans to be individually encapsulated with the addresses of the cash provider and cash taker, and some data structure to hold a credit score for the loan (such as an array of values that indicate for each month whether the payment was repaid, late, defaulted, *etc.*). Note that it is not up to the CreditEvent object to penalize credit events. It passively records them and then protection objects can chose how to act. CreditEvent should be agnostic of what type of loan it is representing (*e.g.,* peer-to-peer, bond, *etc.*). In Ụgwọ, each bond is an individual loan. Protection objects, like credit default swaps, are generally written to monitor credit events across the entire issue of bonds, not just one individual bond. We leave for future work improvements to how sets of loans can be insured.

This credit history could be a struct, mapping or array. According to solidity documentation, in order to restrict the size of a struct, 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 of itself that contains the monthly defaults. In Solidity, mappings are like hash tables that are initialized dynamically with key/value pairs. Unmapped keys return an all zero byte-representation. However, it is not possible to iterate through the contents of a mapping and therefore, the best implementation was to have an array contained within a struct. In all cases, the inner container cannot be visible within the interface of the Ethereum wallet even if the outside container is made public. 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. To make the contract more developer friendly, we use getter functions to reach inside structs and expose the contents to the wallet interface.

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

[6] Blocks 4652926 to 4832685 were mined in December 2017.

variable that floats around the contract that might not necessarily 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. The interface of the Ethereum wallet for the CreditEvent object contains the parameters for identifying each loan on the CreditEvent object. The loan address is used to retrieve this information. The months which have no default are represented with zero and 300000000000000000 wei (0.3 ether) is the default amount for the second month. To pay out this default, any protection object would just need to fetch the value from the CreditEvent object.

### 4.5 Implementing a Credit Default Swap

A way to address counter party risk, without solving it, is to have a third party provide insurance on a loan. Such a contract is both a protection object and also introduces a new counter-party risk: that the insurer will default on paying the insurance if a credit event occurs. We implement a very simple CDS contract. The basic CDS contract is drawn up by the insurance seller who initializes agreed upon facts such as the CDS buyer, amount to be insured, premium, among others. During the payment by the CDS buyer, the function allocates space in the CreditEvent object to hold information regarding the standings of payments made to the CDS buyer.

If a default occurs on a loan that has been insured with a CDS, the default function would be run by the CDS buyer (the buyer has a higher stake and more incentive to run the function). This function would update the CreditEvent object with the balance of the loan to be paid. This is because when a default occurs, 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 cash provider. 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. 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 hard to track as it would have a new address with no relation to the old address. The other way, which we implemented, is to have the same loan contract implemented for the CDS change the owner name. This way the new owner (CDS seller) is tied to the loan contract and anyone who had the address for watching the CDS loan would be aware that a credit swap occurred. The change of ownership is also reflected in the CreditEvent object.

## 5 Evaluation

Our contracts were developed in Remix and tested on Ethereum's test network.

**Table 1.** Cost of running the basic and loan contracts

| Contract | Gas | Ether | USD |
|---|---|---|---|
| *Base System* | | | |
| 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 |
| *Peer to Peer Lending* | | | |
| 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 |
| *Bond* | | | |
| 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 |

### 5.1 Security

Solidity (and Serpent) is notorious for security issues [2,6,8]. 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 opposed to block numbers) are used in our project, our contract is not time dependent and any modification of the time by factor of 900 s 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 changed after deployment. Therefore, the contracts are not susceptible to a transaction ordering attacks.

**Table 2.** Cost of running the protection contracts

| Contract | Gas | Ether | USD |
|---|---|---|---|
| *Collateral* | | | |
| 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 |
| *Credit Default Swap* | | | |
| CDS Contract | 452,035 | 0.009 | $2.58 |
| Monthly Premium | 204,509 | 0.004 | $1.20 |
| Report Default | 61,709 | 0.001 | $0.16 |
| Kill | 27,664 | 0.001 | $0.16 |

In order to test our system for known security bugs, we use a symbolic execution tool called Oyente [8].[7] The tool has been proved in successfully identifying critical security vulnerability, such as a famous incident called the DAO vulnerability. The various APIs used by both contracts were analyzed together simulating the exact same way it would be deployed. None are vulnerable to any of the tests.

### 5.2 Cost

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. Tables 1 and 2 represent the cost of running each smart contract and its functions contained therein on the Ethereum Virtual Machine. 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.

### 5.3 Concluding Remarks

We have present Ụgwọ, an Ethereum implementation of a lending infrastructure. We use the term infrastructure because Ụgwọ is not a single system, but rather a central component (CreditEvent) with two interfaces for an extensible system,

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

where new loan and loan protection techniques can be added. Future work might deploy more exotic bonds or commercial paper arrangements, or other types of protection techniques like reputation systems and repurchase agreements.

**Acknowledgements.** J. Clark acknowledges funding for this work from NSERC and FQRNT.

## References

1. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE Symposium on Security and Privacy (2014)
2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
3. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
4. Danezis, G., Meiklejohn, S.: Centrally banked cryptocurrencies. In: NDSS (2015)
5. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Pelc, A., Schwarzmann, A.A. (eds.) SSS 2015. LNCS, vol. 9212, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21741-3_1
6. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 79–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_6
7. Latter, T.: The choice of exchange rate regime. In: Centre for Central Banking Studies, vol. 2. Bank of England (1996)
8. Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS (2016)
9. Poon, J., Dryja, T.: The bitcoin lightning network: scalable off-chain instant payments. Technical report (draft) (2015). https://lightning.network
10. Ruffing, T., Kate, A., Schröder, D.: Liar, liar, coins on fire!: Penalizing equivocation by loss of Bitcoins. In: CCS (2015)
11. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town crier: an authenticated data feed for smart contracts. In: CCS (2016)