17th May 2018

Distributed Options Exchange on the Ethereum Blockchain

Fredric Moezinia
moezinia@mit.edu

Elliott Forde
eforde@mit.edu

Quinn Magendanz
qpm3@mit.edu

Abstract

As the number of participants in cryptocurrency markets has risen, the founding paradigm of decentralization trustlessness have been lost. And despite increased market size, the level of sophistication of cryptocurrency markets like Ethereum have stagnated.

In an attempt to restore these core principles, and allow complex cryptocurrency derivatives to be traded, we have created a secure and decentralized options exchange. This is known as a Dapp (Distributed Application) which manages an order book and lets users execute orders manually on a website.

The system is distributed and confidential, reducing dependencies on a central servers and exchanges. It is comprised of indefinite and immutable *smart contracts* to enforce all types of contracts in the application.

1 Introduction

Right now, the investment environment around Cryptocurrencies is very limited and insecure. First of all, platforms like Coinbase store your currencies for you, which is both antithetical to the principles of decentralized currencies and a potentially insecure single point of failure. Secondly, it is almost impossible (especially as a retail investor) to hold any non-long position in the market; this has made markets unbalanced.

Additionally, as smart contract code is immutable once committed to the blockchain, the infallibility of smart contract code is of utmost importance. And because smart contracts are a store for sometimes large amounts of money, they are often attacked by hackers. We have seen massive amounts of money stolen from insecure smart contracts. For example, The DAO (distributed autonomous organization) comprised of a series of smart contracts was hacked in June 2016, leaking 15% of all ether in circulation. Because we are creating an application that will be handling potentially large amounts of money, the security aspect is vital.

We intend to create a simple decentralized application for investors to buy and sell Ether options. This flexible type of financial derivative allows two investors to enter into a trade that is mutually beneficial. Due to the nature of smart contracts, the investors are financially bound to the contract.

1.1 Design Principles

Confidentiality: Users are represented by a public key unlinked to any other form of identification. This guarantees the same level of anonymity as Ethereum Wallets.

Integrity: Providing integrity for the user is essential when data is in the form of money. Smart contract code is immutable once deployed to the blockchain. This facilitates consistency and accuracy.

Accessibility: The decentralized nature of the system ensures that contracts will be accessible regardless of access to the system. The interminability of the blockchain and smart contracts ensures availability forever.

Non-Repudiation: The smart contracts hold Ether as collateral, and only distributes at the time of exercising, cancellation or maturity. This ensures that users are held liable for the agreed-upon option.

1.2 Smart Contracts

Smart contracts are the core component of our application. They are similar to object-oriented classes in that they have methods and attributes, and can be instantiated and referred to by a pointer: their address on the Ethereum blockchain. Contracts can also hold and distribute Ether. The functions and structure of the smart contract cannot be changed once written to the blockchain, so it is of utmost importance to ensure its correctness and security.

Calls to smart contract methods are atomic transactions which need to be mined and included in the blockchain to take effect. This means that all method calls cost Ether as miners are providing computation. This system prevents adversaries from attempting denial of service on the Ethereum network. However, currently this small cost is offloaded onto the user, which is not ideal.

2 System Design

The system stack is comprised of a node.js server, a front-end interface, and smart contracts for logic and data storage. Each component will be described in detail below. Once deployed, the smart contracts in our system are unchangeable. On account of this, the security of all components and interfaces is of utmost importance. Buggy contracts can catastrophically render Ether inaccessible or stolen. Our full implementation can be seen at `https://github.com/evforde/eth-options`.

2.1 Node.js Server

The server is hosted on IPFS (InterPlanetary file system [5]) which makes it available for as long as the Ethereum Virtual Machine is running.

The purpose of the server is to provide the client with scripts to render the UI and to interface with the Ethereum blockchain. This, however, does not establish a reliance on the server: all calls to smart contracts may be made manually, without these server scripts. Thus if the server goes down or becomes compromised, users' contracts remain accessible and safe.

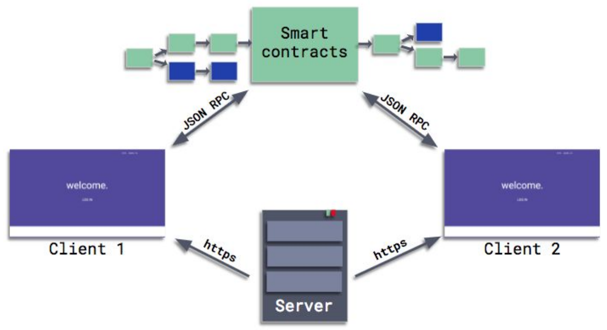


Figure 1: System Diagram

2.2 Client

The Client begins with Metamask, a Chrome browser extension which "includes a secure identity vault, providing a user interface to manage your identities and Ether on different sites and sign blockchain transactions."[1] In the system design, Metamask is used to store a user's public key username, their secret key for signing and encrypting transactions, and some of the users Ether. Further analysis of Metamask reliability is discussed in Section 3.

The front-end interface is a website for users to view their options and trade on the global exchange. Upon login via Metamask, users have access to their personal dashboard and the exchange listings. Users can view their personal ongoing offers and active options in the dashboard, and the user can navigate to the exchange tab to browse and accept open option offers. Users can also make offers for exchange-traded options at specific maturity times and strike prices. The user must have Ether in their Metamask account in order to trade options. This Ether is used as collateral to ensure users are held accountable for the option contracts into which they enter.

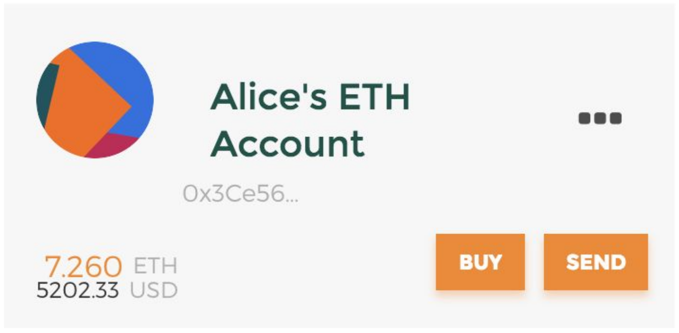


Figure 2: Metamask Extension.

2.3 Smart Contracts

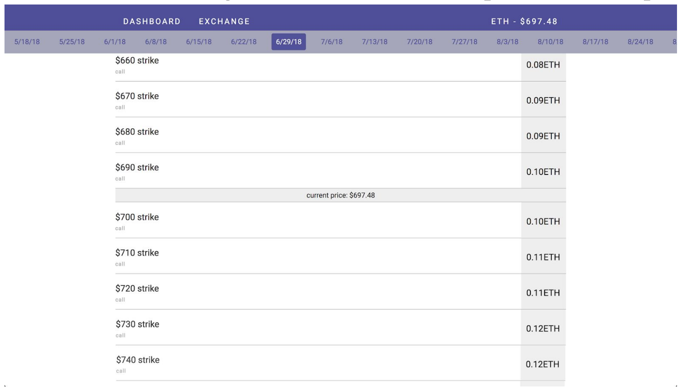
When users create an offer for an exchange traded option, they deploy a new instance of the option smart contract onto the Ethereum blockchain. Then, their orders are publically available for other users of the exchange, who may accept an offer by binding their address to the already deployed contract and sending the required collateral.

To keep track of outstanding orders, we also create an order book contract. This stores the address of all current options in the exchange. The order book holds addresses only of current options, and is used as our database to populate the dashboard and exchange on the front-end.

Fulfilled or active options are stored in cookies in the user's browser to optimize rendering of the UI. We also maintain these active options on the blockchain in case cookies are removed and a user's options need to be restored.

By using the Ethereum blockchain to store option contracts and the order book, we remove the dependency on a central server for data storage. Anyone can read the current state of the exchange by looking at the order book contract on the blockchain. This choice also helps provide data integrity, preventing an adversary from changing option listings.

It is important to recognize, however, that storage on the blockchain is valuable and therefore expensive. Thus, this current design for the exchange order book would scale poorly with the exchange size. An alternative to be considered as the exchange starts to grow at a faster rate would be to store the order book on IPFS. This approach would be distributed and free but increases the implementation complexity. To achieve the desired decentralization and security guarantees, using the blockchain for storage is sufficient for this proof of concept.



$660 strike call	0.08ETH
$670 strike call	0.09ETH
$680 strike call	0.09ETH
$690 strike call	0.10ETH
current price: $697.48	
$700 strike call	0.10ETH
$710 strike call	0.11ETH
$720 strike call	0.11ETH
$730 strike call	0.12ETH
$740 strike call	0.12ETH

Figure 3: List of exchange-traded options.

2.4 Option Structure

In our system, an option is the right to purchase 1 Ether before a certain date (maturity) at a certain price (USD strike). There are two sides, the buyer and seller, who enter into this option contract for making this trade at the specified price (also called the premium). The option contract contains fields for the buyer and seller's public ethereum addresses, the type of option (call or put), the strike price at which the option may be executed, and a cancellation and a maturity time. All these fields and a certain collateral are provided by user who initializes the option when they create an offer. A second user who agrees upon these terms may then accept the option. The contract can be thought of as an escrow held in the blockchain.

We implement physical settlement Options, meaning Ether is transferred to the buyer at the time of exercising. This is because the only currency the system handles is Ether, not USD.

2.4.1 Option Creation and Offer Acceptance

If the user does not find an existing option offer on the exchange which they want to accept, options can be created and offered to all other users on the exchange. When a user creates an option, this instantiates a new smart contract as shown below.

```
constructor(uint _strikePrice,
    uint _maturityT, uint _cancellationT,
    uint _premium, string _traderType,
    address _orderBook) public payable {
  require(_maturityT > block.timestamp);
  require(_cancellationT > block.timestamp);
  require(_cancellationT <= _maturityTime);
  if (traderType == “buyer”) {
    require(valueSent >= premium);
  }
  else {
    require(valueSent >= underlyingAmount);
  }
  _orderBook.addOption(this);
  ...
}
```

Figure 4: Option Instantiation Pseudocode

Based on the option parameters, client side scripts pre calculate how much collateral (`valueSent`) the creating user needs to send to the option smart contract to make the process seamless. The contract validates all attributes of the option as shown in Figure 4 with the require keyword.

When a second user chooses to take the other side of the option, the smart contract again validates the call and collateral before making the option active.

2.4.2 Option Exercising

Options can be exercised by the click of a button in the dashboard. Using a back-end price fetcher, we can indicate to the user which options are in or out of the money. This feature is for convenience only, and the option contract performs its own, independent price fetching and tests to ensure callers do not spoof the price. A call to exercise the smart contract validates the sender’s address and option status before distributing the Ether accordingly.

The buyer receives

$$\frac{currentETHPriceUSD - strikePriceUSD}{currentETHPriceUSD}$$

and the seller receives the remaining collateral,

$$\frac{strikePriceUSD}{currentETHPriceUSD}$$

in Ethereum. Once the parties have received the Ether, the smart contract self destructs to free up space on the blockchain and return the creation cost to the creator.

2.4.3 Fetching Price

Firstly, the price of Ether differs from exchange to exchange as the markets move with supply and demand, which may differ between exchanges. Secondly, using an exchange for price means relying on, and trusting, a third party -- a concept which contradicts our Design Principles. Lastly, blockchains are innately blind to the outside world, and can only make calls to other addresses in the blockchain, not APIs.

The option smart contract partially solves this by using an Ethereum Oracle, a smart contract/web-server service dedicated to making calls to resources outside of the blockchain[2]. In exchange for calls to their service, the oracle requires a small toll from the calling contract. This toll is offloaded onto the user who makes the call to exercise to prevent successive calls from draining the contract balance.

To avoid having a single point of failure, and to distribute trust, the option contract calls several different APIs to interpolate a price. One additional strategy which could be implemented to reduce dependency on a single oracle would be to make the different API calls described above using different oracle services, and similarly interpolating a price.

2.4.4 Option Cancellation and Reclamation

The cancellation and maturity times passed in to the contract on initialization specify conditions under which the contract may be canceled or becomes void. Cancellation time refers to the time when a contract can no longer be accepted by a second user as they are removed from the exchange, which allows users to place a time limit on their offers. This is meant to prevent option creators taking on undue risk in the environment of a volatile market. Options can also be cancelled by users with the click of a button.

Maturity time refers to the expiration of the option. At this point, the smart contract will no longer accept function calls, and if the smart contract is called after this point, the contract will return Ether to the seller and self-destruct to free up blockchain space. Unfortunately smart contracts currently have no means of monitoring themselves and their attributes and can only be triggered by external function calls. This means that contracts cannot automatically void themselves once the cancellation time reached or release funds at maturity, so we place checks in function calls to ensure that we have not exceeded these time limits.

2.5 Communication Protocols

The two channels of communication are between the server and user and user and blockchain. As described in Section 2.2, the server scripts provide a level of abstraction and a user interface to clients to make calling contracts simpler. All packets received for the front-end are sent through HTTPS/SSL, thereby verifying that they are authentic and preventing adversaries from modifying server-sent code.

The communication between users and blockchain happens with the JSON RPC protocol. This is implemented by Web3, the industry standard API used to communicate with the Ethereum blockchain in web applications. All messages sent to smart contracts are cryptographically signed, allowing contracts to verify the identity of message senders.

3 Security Analysis

We will first analyze our security by showing defenses against a handful of major attacks that we kept in mind while building the system. We will also describe a few flaws of our system that we hope to amend in the future.

3.1 Malicious Node.js Server

As noted in the description of the node.js web-server in the system description above, all scripts provided to the client are to make interactions with the options exchange and existing smart contracts intuitive and user-friendly. However, if the server becomes compromised, it contains no user information and no option contract state, so no harmful modifications can be performed. If the server becomes unresponsive and cannot provide scripts at all, the client does not rely upon the user interface given by the server; it it is still possible for the user to manually make calls to their existing option contracts, as well as make calls to the order book contract to obtain the current state of the system and accept/propose new open contracts.

3.2 Spoofed Options Smart Contracts

An adversary may create a custom smart contract with the same interface as an option smart contract that simply forwards all funds to oneself. Such a contract would appear to a user to be a real option contract, but could steal the user's Ether if they tried to accept it. To protect users from these malicious contracts, the order book verifies all new contracts added to the exchange, checking the bytecode (the actual binary instructions that make up the code and structure of a smart contract) of the new option, and verifying that its hash is what we expect of a legitimate option smart contract. Only new smart contracts that have the same hashed bytecode will be added to the exchange. This prevents malicious smart contracts from being viewed in the exchange.

3.3 Smart Contract Timestamps

Both cancellation and reclamation calls to the smart contract are transactions that need to be mined. In these specific transaction operation, the smart contract relies on a current timestamp of the transaction's miner to validate that the call and to cancel or expire the contract. Since the transaction miner provides the timestamp, an adversary could mine their own transaction and provide incorrect timestamps. This attack would allow an adversary to cancel an option early to reclaim their funds or exercise an option after expiry.

This vulnerability, however, is addressed by a passive property of the blockchain: if a block's timestamp is too far in the future, no other blocks will be appended to it. Additionally, blocks cannot have timestamps earlier than their parent block. Thus, in the event that an adversary has the compute power to mine their own malicious transaction, it would become a dead fork in the blockchain, rendering it invalid.

3.4 Metamask Unreliability

Since our initial system design, vulnerabilities in Metamask have been identified which can lead to adversaries leaking private keys or tricking users into transfering Ether to adversary-controlled addresses[6]. We have been especially careful to ensure that cross-site scripting attacks cannot be mounted on our website, so many of these Metamask vulnerabilities cannot be enacted. Alternatively, we may adapt our interface to be more accepting of alternative client wallet providers in the future.

3.5 Reentrant Functions

All functions in the smart contracts need to be sure to set the updated state of the contract before passing control flow out of the current function. Failure to do this may result in an attack known as reentry, where transferred control may continue to withdraw funds from a vulnerable contract that does not update its internal bookkeeping. We have designed our contracts to be secure against reentrant attacks.

4 Future Work

One existing issue with developing smart contracts is that once a smart contract is deployed to the blockchain, its API function calls cannot be edit or changed. Buggy contracts may lock up user funds, allow arbitrary users to drain their funds, or be otherwise vulnerable to attacks. Given the importance of detecting vulnerabilities in smart contract code, we propose that once we have a version of the option smart contract and the order book smart contract which are ready for deployment, we construct a formal proof for both of these contracts to prove correctness of the code. Amazon has used these formal proofs to identify dozens of new bugs across a few of their web systems, and actively use this technique on developing products[3]. An additional bug/vulnerability identification technique which we are looking to apply to the completed distributed options exchange system is symbolic execution. Symbolic execution represents all variables derived from function inputs as symbolic values as opposed to concrete values. This representation allows for the analysis of systems under every possible input space, and reveals overlooked corner cases. For a more in depth description of symbolic execution, see EXE[5].

Additionally, to reduce our dependence upon single third parties for price information, we may explore the design of a system that consumes price information from multiple sources and rewards sources for supplying accurate information. Thus, single malicious suppliers may have no effect on the effective price used in the exchange, and suppliers are motivated to be benevolent.

5 Conclusion

From a financial perspective, our platform is still relatively simple. The underlying technical aspects present real value in the realm of decentralized and trustless systems. The system's UI and APIs can easily be expanded to include more types of derivatives and even currencies other than Ethereum.

References

1. https://metamask.io/
2. http://www.oraclize.it/
3. Use of Formal Methods at Amazon Web Services (http://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf)
4. EXE: Automatically Generating Inputs of Death (https://web.stanford.edu/~engler/exe-ccs-06.pdf)
5. https://ipfs.io/
6. 6.857 Final Project: Zhang, Shao, Chang, Hao