EasyChair Preprint

№ 7

On the feasibility of decentralized derivatives markets

Shayan Eskandari, Jeremy Clark, Moe Adham and Vignesh Sundaresan

February 13, 2018

On the feasibility of decentralized derivatives markets

Shayan Eskandari[1] Jeremy Clark[2] Vignesh Sundaresan[1] Moe Adham[1]

1 Bitaccess
2 Concordia University

Abstract. In this paper, we present Velocity, a decentralized market deployed on Ethereum for trading a custom type of derivative option. To enable the smart contract to work, we also implement a price fetching tool called PriceGeth. We present this as a case study, noting challenges in development of the system that might be of independent interest to whose working on smart contract implementations. We also apply recent academic results on the security of the Solidity smart contract language in validating our code's security. Finally, we discuss more generally the use of smart contracts in modelling financial derivatives.

1 Introductory Remarks

The introduction of Bitcoin [13] in 2009 led to a new frontier in decentralizing technologies, both in finance and elsewhere. Of the many implementations, we note a few: file systems like The InterPlanetary File System (IPFS) [2], dynamic name servers like DNSChain [14] and MaidSafe, a fully distributed platform [10]. For our purposes, the most interesting technology is Ethereum [4][17] — a decentralized general transaction ledger. Ethereum in simple words is a decentralized computer that can run code, called smart contracts, which enforce the performance of an agreed upon set of negotiated standards in an automated and immutable way. Smart contracts can be designed to disintermediate traditional trusted parties, replacing them with pre-defined logical parameters. The smart contract concept is not new and was introduced by Szabo in 1997 [16], however there has not been any real implementation of it until Bitcoin, and then in a much more flexible and verbose fashion: Ethereum.

Under the umbrella of "fintech", "blockchain", and "distributed ledger technology", many legacy entities in the financial world (investment banks, security exchanges, clearinghouses, etc.) have expressed interest (through whitepapers and commercial partnerships and consortiums) in decentralizing financial markets. Derivative markets are often cited as a potential target. From the other end, papers on Ethereum and tutorials on Solidity (a high level programming language for Ethereum) often use derivatives as an example application. So there is a degree of consensus that derivatives running on Ethereum is an interesting application to study, but we are not aware of any public projects to attempt to build a derivative market in a serious way. This paper is a first step in that direction.

1.1 Scope & Contributions.

A simplification of a derivative is as follows: two parties enter an agreement where the first stands to profit if a specified security (*e.g.,* stock) appreciates in value over a specified time-period and the second stands to profit if it falls. Since the profitability of the agreement is derived directly from the price of the security, it is called a derivative instrument. The exact operational details that realize this property differs between types of derivatives. The most common derivative is a put/call option which gives the second party (called the *buyer*) the opportunity (but not obligation) to buy/sell a security at a specified price (*strike price*) at (*American*) or within (*European*) a specified time (*expiration*). The buyer pays the first party (the *seller*) a flat fee (*option price*) when purchasing the option. Derivatives are generally held to hedge risks in price movements or for speculation.

In a decentralized derivative system, a buyer and seller can have fast and automatic clearing and settlement (straight through processing) of the derivative without trusting a third party. However the design of a market must consider the following challenges:

1. **Terms of the Contract.** The terms of derivative must be expressible in the smart contract language. In this paper, we write contracts in Solidity for the Ethereum blockchain which is sufficient for describing the core aspects of the contract. We present a full implementation stack (from the smart contracts to a UI) for buying/selling a special type of derivative instrument. We pay special attention to common security risks in developing Solidity-based contracts.
2. **Counterparty Risk.** In most derivatives, the seller is obliged to buy/sell securities upon request of the buyer subject to the terms of the derivative. A seller might choose to not follow through with her obligations. In a centralized setting, identity, reputation and legal recourse are used to combat this. In a decentralized environment, this problem must be addressed. In this paper (and the reason we position it as a first step), we start with derivatives that are fully collateralized — meaning the full settlement amount under all outcomes is capped and this amount is locked to the contract at initiation time and distributed under the conditions of the contract. This means we do not implement a traditional put/call option but rather a tweaked version we describe below. In future work, we will consider counterparty risk broadly and how mitigating it can be combined with our framework to offer more traditional derivatives.
3. **Price Feed.** In a derivative where settlement is fully automated, either the underlying security (or a token representing it) needs to be on the blockchain already or the blockchain needs to be able to assign a value to the security— or more precisely, be fed the price it should use in evaluating the code of the contract. In practice, an entity feeding prices (or any external information) into a smart contract is called an oracle. Some related work has examined

oracles, and we present our decentralized design in Section subsection 4.2 called PriceGeth, which we have made freely available.[1]

4. **Underlying Financial Model.** The buyer and seller of a derivative, whether implicitly or explicitly, must have some sense of what the probabilistic behaviour of the underlying security must be to determine the terms of the contract. This is the purpose of the infamous Nobel-awarded Black-Scholes model for stock prices — now obsolete but influential for decades. In our system, such a model is not baked into the functioning of the smart contract but would be used externally to decide favourable terms before buying/selling derivatives. For stocks, modern models (like jump-diffusion) might be used. For derivatives on cryptocurrencies or more esoteric securities, models simply do not exist yet and are an open area of research. Finally, we note that the derivative ultimately settles in Ether and so inflations/deflation of the currency might erode an otherwise profitable derivative.

In summary, we limit our contributions to (1) and (3) in this work, but also propose this fuller landscape as a useful research agenda for future researchers.

2 Related Work

Work on trusted oracles and price feeds, in the Ethereum eco-system, include TownCrier [18] which acts as an attested bridge (running within an SGX enclave) between trusted sources of information and the Ethereum blockchain. Oraclizeit[2] is another price feed which uses the similar workflow to fetch the requested information. Our approach differs from these as PriceGeth publishes the data to the Ethereum blockchain from the trusted source of information and the historical data is available to all smart contracts, however in comparison with the other approaches, is limited to only the published data (Price pairs).

Equibit [11] proposes a method to issue, create, disseminate and maintain equity across a broad base of investors without the need of intermediaries for record keeping. It is conceivable that derivative smart-contracts could utilize Equibit equity as payment or settlement method, as opposed to simply using Bitcoin or Ethereum's native digital currencies.

Bentov et al. [3] note than an extension to their work on decentralized prediction markets can be a derivative instruments they call a *capped contracts for difference*. It is similar to the one implemented in Velocity (their paper is not an implementation but a study of game theoretic properties).

Recent attacks on smart-contracts, such as TheDAO attack [9] attracted security researchers to analyze further on this era. Solidity security and survey of the attacks by Atzei et al. [1] lists some of the known security vulnerabilities and Luu et al. developed a tool for static analysis on smart contract codes [12] which we used.

[1] `https://github.com/VelocityMarket/pricegeth`
[2] `http://www.oraclize.it/`

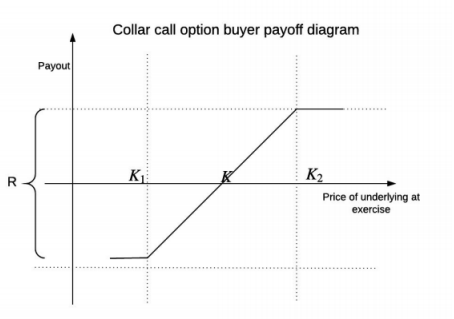

Fig. 1: Our collar-esque option with maximum long payout scenario. K1 is the initial price, K2 is the price at expiry time and R is the pre-defined collar for payouts

3 Materials and Methods

Smart Contracts. A *contract* is a written or spoken agreement between two or more parties that is intended to be enforceable by law. In a *smart contract*, terms are written in code and executed by machines, removing the human performance component (unless if such a component is specified). We can consider our main smart contract as a black box: the inputs are investors' deposited *ether* (Ethereum's cash) and their position on the future price of an asset, either short or long. The smart contract will retain the deposit in escrow and execute a payout calculation and the payout itself when the expiry date comes. The payout is in Ether only, no actual shares are exchanged (a *contract for difference*) and the maximum payout is capped (*limit up/down*). Due to the deposit, there is no counter-party risk however the contract requires a trustworthy price feed and the investors earn zero interest for the duration of the contract. For this reason, we consider this a first step toward more flexible arrangements. The contract disintermediates the trusted role of the exchange (or broker for over-the-counter) and settling/clearing entities.

Types of Options. We implement a non-standard option that is similar to a collar or hedge wrapper. It is non-standard due to our requirement of escrowing money, which we make to side-step counter-party risk and enable a fully autonomous and disintermediated contract. The contract collects funds from the hedgers/speculators who take opposing positions on the future prospects of an asset: one takes the short position when they believe the underlying asset's value will lose value from its current price, and other takes the opposite long position speculating a rise in the price. In its simplest form, the collar options pay out \$1 for every \$1 change in the underlying asset (the payout can be made dependent on a drift term or even made non-linear). The payout is limited by the amount of money held in escrow—if the price rises beyond the limit, it is said to be limit up (or limit down in the opposite case) and the payout will be fixed (see

Figure 1). This kind of payout capping helps the contract holders stay immune to systemic risks and extreme jumps.

Development and Deployment. There are a few blockchains that would let us code an autonomous smart contracts: Ethereum, RSK [8] and more. The decision to work on Ethereum blockchain rather than others solely came from the fact that there are more active developers in the community and maturity of the platform. Even though Ethereum is in early stages, it is more mature than other smart contract compatible platforms. The programming language used for smart contract development is Solidity in most of these platforms. All smart contracts developed and used in this paper has been deployed and tested by our beta testers on Ethereum testnet. In Ethereum blockchain, transactions and processing power costs some small amount of ether called *gas*[3]. For each transaction, the sender defines the gasLimit and also gasPrice for processing that transaction and miners decide to include those transactions in the blocks they mine or not. The concept of gas has many angles to discuss which falls outside of the scope of this paper. We will discuss some more in section 5.

4 Implementation

We call our platform Velocity. We tried to model the real-life scenario of buying an options derivatives. Consider the case where Alice goes to a broker and buys an options contract from Bob. The broker is the one that handles the money transfer and also execute the options contract at the contract expiry time. Now our goal is to replace the broker with a smart contract. For the purpose of a proof of concept, **the smart contract will also act as Bob**, meaning if Alice buys a short call option, the Velocity smart contract will put a long call against her short call. This can be generalized so that other entities can fund the contract but for the rest of this paper, Velocity acts as a market maker. This might lead to users gaming the system, however it's trivial to change the smart contract to wait for the other opponent to enter the contract. We discuss this more in section 5.

4.1 Velocity Main Smart Contract

A Velocity smart contract can be used for speculation on the price of any two assets[4], although the Ethereum price is always exposed as the deposits and the withdrawals are done in ETH[5]. As for this experiment, we use the price pair of Bitcoin (XBT/BTC) and Ethereum (ETH). If we used price pairs not involving ETH, for example the CAD/USD exchange rate, it would suffice to use two contracts for CAD/ETH and ETH/USD. Or the payout function could be changed to specify how it relates to numbers it is given. Note that in either case,

[3] What is gas? `http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html#what-is-gas`

[4] or any other events that an options contract can be based on

[5] Ethereum symbol

the payout will always be in ETH. In its full generality, any number that changes over time and has a suitable feed (we describe feeds below) can be used: price (stocks, bonds, commodities, etc.), rate (interest, inflation, population, etc.), or something else (average global temperature, number of days without rain, etc).

Smart contract. The way Velocity smart contract is implemented, one party purchases a contract by sending a nominal amount of ethereum (0.1 ETH) to the contract's ethereum address. Once confirmed by the network, the contract will fetch a starting price from the price feed, PriceGeth, and run for a period of time to reach the expiry time. The smart contract would put the same amount of ETH from its pool of funds into escrow for the payout. In the PoC demo, we use 5 ethereum blocks (approximately 1 minute) to settle a contract. When the expiry time reaches, the same party must send another transaction to the contract and call the settlement function to settle the contract which leads to sending the payouts by the smart contract. While this experiment was going under beta testings, we found out that if the user loses the contract, there is no incentive to call the settle function as it would use up some ETH in gas and would not pay the user. This would lead stale money held in the escrow of the smart contract. This made us redesign our settlement functions and write one centralized cron job script to go through the unsettled contracts once a day and call the settle function on the ones that have been expired.

```
1  modifier checkMargin(uint amount) {
2     if (amount == (applyLOT(Margin)))
3     { _ ;} else {
4         Error("Invalid Margin!");
5         immediateRefund();}
6     }
7  function goLong() public hasEnoughFunds(msg.value) checkMargin(msg.value)
   ↪  payable returns(uint) {
8     lastOptionId = newOption(msg.sender, msg.value, true);
9     LongOption(lastOptionId, msg.sender, msg.value, block.number);
10    return lastOptionId;
11    }
```

Code 1: Velocity Main Smart Contract - Long Option Call, The sender of a transaction to goLong() function has to send exactly the Margin value and with that he enters the option contract for Margin value with the smart Contract

Settle function. exercise() is responsible in settling the options contract and pay out both parties (see 2), in which here is the user and the Velocity smart

contract. Most of the functions are responsible to find the appropriate option contract and calculate the pay outs. However there are some functions that were added later on for security measurements, such as isOpen modifier. Modifiers in Solidity are functions that can check some statements before executing the main function. The first deployed version of Velocity main contract was vulnerable to a similar (but not the same) attack as the DAO attack, see section 5. It was possible for an attacker to call an option contract and upon settling and winning, keep calling the exercise() function using his OptionId and get more of the same amount of payout over and over again. The code was patched and a new smart contract was deployed later in the experiment[6]. send() is a built-in function in Solidity which handles the sending of funds to other ethereum addresses or contracts. There are known vulnerabilities on how send() function works in solidity which should be appropriately handled. One can use a smart contract address as his option payout address which would execute some code upon receiving any funds and use that code flow to drain the sender's contract. payAndHandle() function tried to use the best security practices to prevent such attacks (see 5 for the source code).

```
1   modifier isOpen(uint optionId) {if (AllOptions[optionId].closed) throw;
    ↪ _ ;}
2   function exercise() public {
3     exercise(findOptionId(msg.sender));
4   }
5   function exercise(uint optionId) public isOpen(optionId) returns(bool)
    ↪ {
6     // REMOVED SOME CODE TO SAVE SPACE, FULL SOURCE CODE IS AVAILABLE ON
    ↪ VELOCITY GITHUB REPOSITORY
7     AllOptions[optionId].closed = true; //Doing this before payouts to
    ↪ prevent replay attacks on same instance of the contract
8     LockedBalance -= AllOptions[optionId].amount; //release locked amount
    ↪ from escrow
9     // Payout calculation
10    if (pricesToCheck.pricediff >= (int(Margin))) { // diff >= (margin)
    ↪ -> Pay Long
11        //pay long
12        return payAndHandle(optionId, AllOptions[optionId].Long, 2 *
    ↪ AllOptions[optionId].amount);
13    }
14    if ((0 < pricesToCheck.pricediff) && (pricesToCheck.pricediff <
    ↪ (int(Margin)))) { // 0 < diff <  margin
```

[6] Fix for the multiple payout bug: https://github.com/VelocityMarket/Options-Contract/commit/f3c8d0ef66b886c9ee8b432e92c83f3a4fb525ba

```
15        return (payAndHandle(optionId, AllOptions[optionId].Long,
    ↪  (AllOptions[optionId].amount + pricesToCheck.priceDiffLOT)) &&
    ↪  payAndHandle(optionId, AllOptions[optionId].Short,
    ↪  (AllOptions[optionId].amount - pricesToCheck.priceDiffLOT)));
16      }
17    }
```

Code 2: Settle function of main options contract

Source Code API documentation for other smart contracts to use the functionality and also Python and NodeJS clients to communicate with the main smart contract are available on Github[7].

4.2 Price feed

A decentralized Price feed is an essential requirement for having a decentralized derivative market. There are a few proposals on how to fetch the price in a smart contract. One is using *Smart Contract* oracles[8], they offer daily updates for the price using a predefined data source. This was not an option to be used for our purpose as a daily update is not sufficient for short term derivative markets. Another option that could be used was Oraclizeit. They way Oraclizeit works is that the client smart contract, Velocity main contract in our case, sends a transaction to Oraclizeit smart contract with the required API url and the fields it needs, sometime after the confirmation by the network, Oraclizeit smart contract sends a callback transaction to Velocity smart contract with the requested data (Figure 2).

For the first implementation of Velocity smart contract we used Oraclizeit method to fetch the price.

As mentioned before, most of the decentralized application infrastructure on Ethereum blockchain are in Beta state and might not work as intended. This applies for Oraclizeit, specially as by design they have a central server which can stop working without any notice or visible signs. The red boxes in Figure 2 indicates the centralized parts of the system. As you can see in 3, Oraclizeit will send the price to the callback function at the time of the call and also execute the exercise() function which is responsible for saving the price and calculating the payout amounts. This makes the callback function one of the important functions which should be called at the specific time.

```
1  //initiating oraclize it
2  oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);
```

[7] Simple collared option smart contract: `https://github.com/VelocityMarket/Options-Contract`

[8] Data and Payments for your Smart Contracts `https://smartcontract.com/`

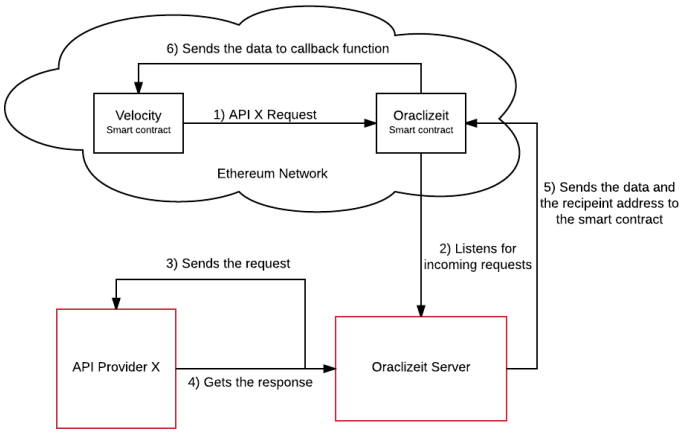

Fig. 2: Oraclizeit work flow

```
//oraclize_setNetwork(2); //
priceUrl = "json(https://www.bitstamp.net/api/v2/ticker/btcusd).last";
function updateBTCUSDFromFeed(uint delay){
  oraclize_query(delay, "URL",
    priceUrl, 400000);
  }
function __callback(bytes32 myid, string result, bytes proof) {
  if (msg.sender != oraclize_cbAddress()) throw;
    uint BTCUSDFeed;
    BTCUSDFeed = parseInt(result, 2);
  exercise() // this function exercises the contract to calculate the
  ↪  payouts
  }
```

Code 3: Implementation of Oraclizeit price feed in Velocity smart contract

In our testing period, we encountered multiple problems with this design:

1. The callback would not happen at all, which would result in an unsettled options contract. Oraclizeit support team were helpful and fixed this issue later on.
2. The callback would happen with some delays, which would result in inconsistency in the fetched price with the the options contract expiry date. decentralized networks have some latency by design, realtime does not really

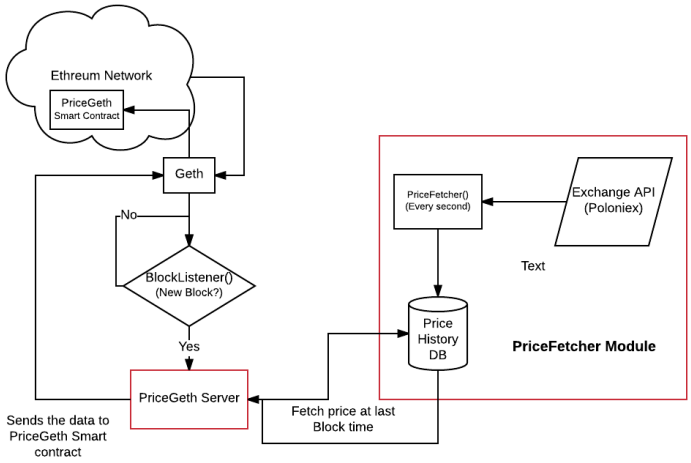

Fig. 3: PriceGeth Work Flow

mean anything in such networks, hence counting on a transaction to happen at a exact time is not the best solution.

3. The callback would happen with insufficient gas, which would result in the failure to properly run exercise() function and thus failiure to settle the options contract. Oraclizeit library offers a way to send more gas than needed in case the callback function needs more gas, however on the time of this experiment that functionality was not working properly.

PriceGeth We designed PriceGeth[9] to publish (almost) realtime price pairs to Ethereum blockchain. This is how PriceGeth works (also see Figure 3):

1. PriceFetcher server is saving an exchange Prices (USDBTC, BTCETH, BTCETC, BTCDOGE) every 1 second in a database
2. BlockListener is listening on using Geth[10] for new blocks
3. When BlockListener sees a new block it fetches the price at the Blocktime from PriceFetcher Module
4. PriceGeth server sends the data to PriceGeth smart contract(4) and updates the latest price

PriceGeth smart contract would keep all the historical prices and all would be available to all smart contracts on Ethereum blockchain for free (no gas needed to fetch the price). The reason this is almost realtime, goes back to the nature of blockchains. Time units as in seconds and minutes are not meaningful for most of the blockchain applications, but the block height can be used as the time

[9] Price API for Smart-Contracts on Ethereum Blockchain `https://github.com/VelocityMarket/pricegeth`

[10] Official Go implementation of the Ethereum protocol `https://geth.ethereum.org`

unit, meaning the time of each block is known to all users of the blockchain, but before a block is published no other time units can be used. This is why we designed PriceFetcher module to connect to an exchange API and saves the price pairs every second, to have the price for the previous block time anytime a new Ethereum block is generated.

```
struct Feed {
      uint    USDBTC;
      uint40    BTCETH;
      uint40    BTCETC;
      uint40    BTCDOGE;
      uint40  timestamp;
      uint    blockNumber;
    }
mapping (uint => Feed) priceHistory;
function setPrice(uint40 timestamp, uint40 blocknumber, uint USDBTC,
 ↪  uint40 BTCETH, uint40 BTCETC, uint40 BTCDOGE) ifOwner() {
  if (firstBlock == 0) firstBlock = blocknumber;
  priceHistory[lastBlock].timestamp = timestamp;
  priceHistory[lastBlock].blockNumber = blocknumber;
  priceHistory[lastBlock].USDBTC = USDBTC;
  priceHistory[lastBlock].BTCETH = BTCETH;
  priceHistory[lastBlock].BTCETC = BTCETC;
  priceHistory[lastBlock].BTCDOGE = BTCDOGE;
  PriceUpdated(timestamp, blocknumber, USDBTC, BTCETH, BTCETC, BTCDOGE);
}
```

Code 4: Pricegeth Main Smart contract

PriceGeth is a proof of concept implementation of having a trusted entity publishing price pairs to the blockchain and we are aware of the implications of trusting the PriceFetcher not to manipulate the prices. PriceFetcher is the central point of failure in PriceGeth design and should be addressed in future work. However after further research, it is almost impossible to have a truly trustless decentralized price feed unless we have a decentralized exchange infrastructure on the blockchain. This exchange can be used as the price oracle as the order books would be stored on the blockchain and hence there is no one single point of trust. The red boxes in Figure 3 are indicating the centralized parts of this implementation. PriceGeth is released as a stand alone smart contract and also a library to be used in other smart contracts to use the price feed free of charge[11]. Another challenge of PriceGeth design is that PricePublisher is paying the gas

[11] PriceGeth Library `https://github.com/VelocityMarket/pricegeth`

for publishing and storing all the price pairs, and as there is no incentive of doing so, it is not an inefficient way of offering price oracles. PriceGeth can be implemented in a way that clients should use a token issued to them beforehand to fetch the price, or require payments to release the price data.

By design PriceGeth operator should not be able to use Velocity options as he can manipulate the price to game the system.

There is a similar work on price feeds titled Town Crier [18], which uses TLS security to prove the fact that the data sent to the smart contract is exactly as the one provided by the API, conceptually similar to Oraclizeit TLSNotary-proof[12]. TownCrier uses Intel SGX in their central server which insures the integrity of hardware used and thus insures no manipulation is done on the server. Even though one can argue that the data provider is a trusted entity, one of the goals to have a decentralized application is to have no trusted entity in the infrastructure and to have a trustless system.

5 Discussion

Security Smart contracts have introduced some new security concerns to developers. Notions like gas usage and consensus and most importantly a function that pays out irreversible money are new to most of the developers hence the ability to develop a secure smart contract is hard to grasp. One of the visible examples of security issues is the attack on The DAO, Decentralized Autonomous Organization[13]. The goal of the DAO was to remove all the need for any venture capital intervention or any other third party for fundraising on a new idea or a company through crowdfunding and giving the investors tokens (shares) of the company. However due to an issue splitDAO function which was responsible to manage and fund new child DAOs or projects, an attacker was able to take one third of the money in the original DAO, worth approximately 86 million USD [7] at the time of the attack, this vulnerability is dubbed *Reentrancy Vulnerability*.

Luu et al. [12] developed a symbolic execution tool called "Oyente" to find potential security bugs, which they proved effective by running on Ethereum blockchain and successfully identifying The DAO vulnerability. We used this tool to analyze our code (see Figure 4).

Another family of vulnerabilities that have caused some of the known attacks are *Mishandled Exceptions*, which mostly has caused Denial of Service attacks on individual smart contracts. In Velocity main contract we used *modifier* functions to sanitize the inputs to narrow down the probability of such exceptions. Another set of attacks *Timestamp Dependence* and *Transaction-Ordering Dependence* are interesting to ponder, however due to the design of Velocity and PriceGeth, they are not applicable to these smart contracts. As an example, usage of timestamp was replaced by Ethereum blocknumber and smart contracts time is based on the block number rather than seconds and minutes. There has been more security bugs in solidity compiler, a few related bugs were explained in 4.1.

[12] `https://docs.oraclize.it/#security-tlsnotary-proof`
[13] `https://github.com/slockit/DAO`

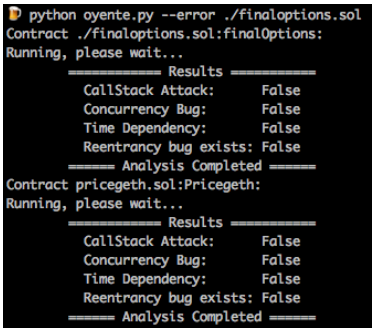

Fig. 4: Results of Smart Contract analysis tool called Oyente [12] to find security bugs

```
1     function payAndHandle(uint optionId, address addr, uint amount)
      ↪ private returns (bool success) {
2       if (addr.send(amount)) {
3             optionPaid(optionId, addr, amount); //event for successful
      ↪ payment
4       } else { throw;}
5       return true;
6     }
```

Code 5: Secure payouts in smart contracts

Gas Sustainability The concept of gas usage for processing power is not easy to grasp even for long term developers. People might be familiar with limited computational or storage resources, but the concept of passing gasLimit to a function to use to process inputs is a new concept. Each step has its own estimated gas usage, as an example to store a value in a variable, you have to pay *100 Wei*[14] for each *sstore* call[15]. This should be considered that there's a cap for gas usage for each transaction and block, thus complex computation should be split into multiple transactions which makes smart contract design more complicated than they are. Also we should mention that function calls can fail due to the fact that they run out of gas and they don't have enough gas to finish their required computation or storage. This can cause unpredicted behaviour from the smart contract as there would be broken flows in the code which should have been handled by the developer. The gas usage could change as there are updates and security patches to Ethereum protocol, e.g transaction spam attack[16]. It

[14] Wei: Smallest unit of Ethereum, equevalent to 0.000000000000000001 ETH
[15] put into permanent storage
[16] Long-term gas cost changes for IO-heavy operations to mitigate transaction spam attacks `https://github.com/ethereum/EIPs/issues/150`

might take multiple implementation of the same function to find an equilibrium between readability and gas efficiency.

Misuse of the contract In the current implementation of Velocity smart contract, one can call the Long option when he is sure of the price increase between the start time and expiry time and keep on doing this until there is no money left in the smart contract's pool of funds. This is because the smart contract calls the opposite of the incoming option call blindly. However in future work, there should be market scoring rule which depends on how many short option calls are placed comparing to the long calls and make it more expensive to call short when there are more short option calls than long calls.

Collar Option library Velocity smart contract can be used as a module in any other smart contract to handle option calls and execute some functions on the expiry time. This smart contract was written as a proof of concept and was released under *GPL* license[17].

6 Future work

As discussed in subsection 4.2, fully decentralized Price feeds and oracles are needed in order to have a trustless decentralized financial market. This can be done by having a decentralized exchange to extract prices from using smart contracts. Even though there has been many price feed methods discussed, none of them seem to have trustless infrastructure. Smart contracts security is not well practiced and there are many unknown attack vectors in the eco system, from solidity compiler security bugs [15] to best practice security implementations [6], there is work to be done and tests to have a more mature secure eco-system to work with, Specially if the end goal is to have a decentralized financial application in place where money is at stake.

As for the options contracts, there should be more research and work on the payouts to make them smarter. One proposed solution is to have market scoring rules in place, which means if there are more open short option calls than long calls, it should get more expensive to call short options and vice-versa. Smart contracts are unchangeable piece of code that run autonomously, meaning if there's a market crash or systematic error, there cannot be anything to do to suspend the payouts and shut down the application, unless with pre-defined functions in the smart contract which only the owner can trigger, which would be a double standard in the trustless eco-system.

7 Conclusion

Even though the idea of having a fully autonomous and decentralized derivative market is intriguing, the infrastructure to reach this goal is still missing from the

[17] `https://github.com/VelocityMarket/Options-Contract`

underlying network. As for example, price feed is one of the essentials of such a market and it should be done in a fully decentralized trustless way to prevent fraud and market manipulation by the feed provider. All the existing solutions today, have a central point that can manipulate data, it is either the exchange API or the component responsible to publish the price. As discussed before, one of the only solutions to this problem is to have a fully decentralized exchange on the network to provide realtime price feed for other smart contracts. There are some work done on decentralized exchanges [5], although there is no real world deployment of such a system at the time of writing. Smart contracts are fascinating idea that can revolutionize the technology by removing the middlemen, however the underlying technology is more on the proof of concept level than mature enough to be used on the real world scenarios. We should also mention that the barrier for people to have the relevant crypto-currency to work with such systems still exists.

References

1. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts. In *POST*, 2017.
2. J. Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv:1407.3561*, 2014.
3. I. Bentov, A. Mizrahi, and M. Rosenfeld. Decentralized prediction market without arbiters. *arXiv:1701.08421*, 2017.
4. V. Buterin et al. A next-generation smart contract and decentralized application platform, 2014.
5. J. Clark, J. Bonneau, E. W. Felten, J. A. Kroll, A. Miller, and A. Narayanan. On decentralizing prediction markets and order books. In *WEIS*, 2014.
6. ConsenSys. Ethereum contract security techniques and tips. *ConsenSys*, 2016.
7. P. Daian. Analysis of the dao exploit. *Hacking, Distributed*, 2016.
8. S. Demian Lerner. Rootstock: Bitcoin powered smart contracts. *Whitepaper*, 2015.
9. K. Finley. A 50 million dollar hack just showed that the dao was all too human. *wired*, 2016.
10. D. Irvine. Maidsafe distributed file system, 2010.
11. B. Kievit-Kylar, C. Horlacher, M. Godard, and C. Saucier. Equibit: A peer-to-peer electronic equity system. *arXiv:1612.06953*, 2016.
12. L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, 2016.
13. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
14. okturtles. A blockchain-based dns, http server that fixes https security, 2014.
15. C. Reitwiessner. Security alert: Solidity variables can be overwritten in storage. *Ethereum Blog*, 2016.
16. N. Szabo. The idea of smart contracts, 1997.
17. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
18. F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *CCS*, 2016.

A Demo Website (UI) for the Velocity smart contract

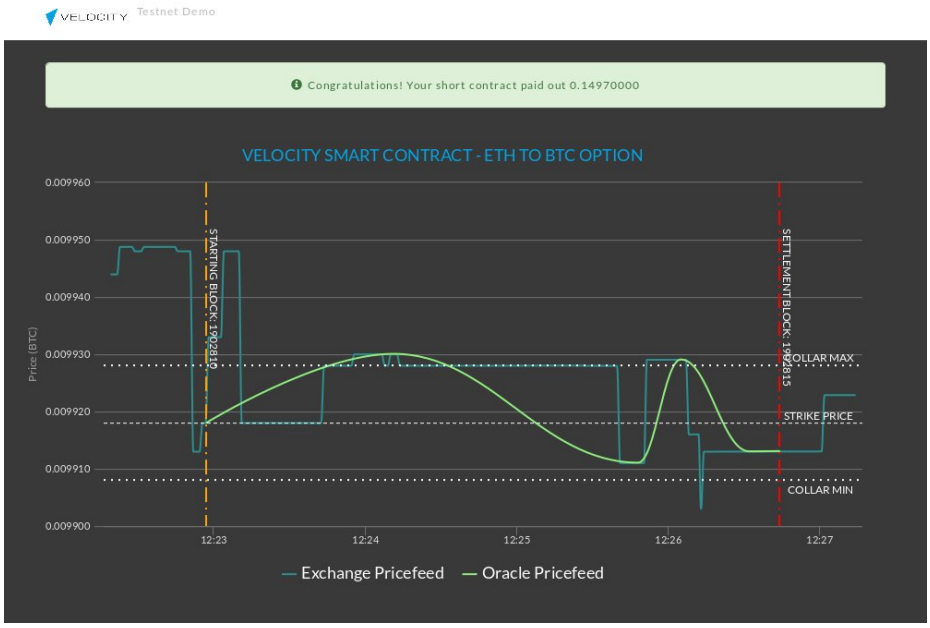

Fig. 5: Velocity Options Smart Contract Demo