Leverj Decentralized Custody

Bharath Rao, Nirmal Gupta

https://leverj.io

Version 0.1

Abstract—**While centralized exchanges are very responsive, they are highly susceptible to custodial risk. Fully decentralized exchanges (DEX) suffer from high latency and high cost, making them impractical for high volume use. Using the on-chain order books and order matching introduce a variety of performance and economic issues that make current DEX unusable. Leverj resolves the trade-offs between centralized and decentralized options by decentralizing safety-critical functions such as custody and centralizing the speed-critical functions such as order matching. Custodial decentralization is done cheaply taking into account blockchain limitations such as congestion and probabilistic finality.**

I. INTRODUCTION

Bitcoin[1] was introduced as a peer-to-peer electronic cash system. Today there are hundreds of blockchains with varying properties. A blockchain is a system whose rules cannot be broken, which makes it an attractive proposition to all manner of financial applications. This has led to numerous financial applications being rewritten to use the blockchain to employ decentralized models to replace legacy financial systems. These applications attempt to incorporate the blockchains elimination of the trusted third party problem. However, attempts to use blockchains for high-speed applications have produced low adoption. The primary use case for blockchain is trading, which is impractical on high-latency systems due to the following issues:

A. Probabilistic Finality

Proof-of-work means that the longest chain cannot always be predicted and occasional re-organization of the blockchain could cause a transaction's confirmation in a single block to be undone if a competing chain tip grows into a longer chain[2]. Six blocks of confirmations are recommended in Bitcoin before the payment is deemed sufficiently finalized. In practice, frequent smaller payments could do with fewer confirmations and only fewer larger payments need to wait for higher number of confirmations. For the use case of electronic cash payments, reorganization risk can be managed by waiting for sufficient time until the risk of finality is dramatically reduced[3]. Faster block times have more active chain tips[4], which means that there is insufficient certainty of an order's fill or cancelled status, eliminating effective arbitrage or hedging.

B. Latency and High-speed use cases

Requiring an operation to confirm on the blockchain introduces variable latency, which makes creating any predictable system using it quite challenging. For example, user feedback when placing an order or matching two orders and taking them off the book can have long delays making the user experience cumbersome[5]. Trading is a high-speed activity and the ability to cancel an order in milliseconds is essential for a healthy liquid market. In a fast moving market, limit orders would need to chase the price and the user is forced to offer an unreasonable price to get in. During high congestion, the market can be manipulated by overloading the network and creating an arbitrage on the stale price in the blockchain versus the more recent price on a centralized system.[6]

C. Economic pressure and low liquidity

Smart contracts not only produce information at block speed, they also consume information at block speed. This means that any smart contract will always be acting on stale data. The losses due to acting on stale information are cumulative and high volume traders will always prefer to trade on more responsive systems. This results in an economic pressure that drives volume out of DEX into centralized exchanges resulting in nearly every DEX retaining only marginal volume.

II. RELATED WORK

Bitshares[7] created one of the earliest on-chain trading ecosystems with full on-chain trading. All operations were on-chain including order placement, cancellation and executions. Bitshares enjoyed moderate success in its day and continues to have marginal traction as of this writing. However, even a three-second block time was insufficient for widespread adoption. In addition to the issues described above, network effects prevent trading of assets of larger networks such as Ethereum on specialized trading block chains. With a better understanding of the problems of on-chain order books, newer trading systems move order books off-chain and only retain execution on-chain. All on-chain execution systems continue to be susceptible to manipulation and congestion attacks.

III. ON-CHAIN EXCHANGE ATTACKS

Trading requires low latency and deterministic finality. On-chain attacks simply take advantage of high-latency and probabilistic finality. All attacks are economically viable at present on most chains and could potentially drive an on-chain execution exchange to unviability.

A. *Congestion Attack*

An attacker can create or amplify any network congestion cheaply by sending coins or tokens among his own addresses for very low transaction fees/gas.[8] These transactions are attack transactions and do not need to ever confirm; their only job is to clog the network. Any on-chain exchange may simply have to stop all activity. Any market making or other trading bots will simply not be economical due to high gas costs.

B. *Front-running Attack*

When an asset market moves fast in a centralized exchange, it may be economically viable to pay higher gas price and get a fill on your order on the on-chain exchange. This enables an attacker to steal someone else's fill within a timespan of one block. Success rate of this attack can also be amplified by congestion attack or miner collusion[6].

C. *Stale Oracle Arbitrage Attack*

Any smart contract that depends on an oracle for a price feed to settle trades will always be trading on stale prices compared to the centralized exchanges due to block confirm latency. This makes it viable to simply enter at the stale price on the smart contract and exit at the fresher price on a centralized exchange[9]. This arbitrage attack will quickly deplete reserves of such exchanges during big moves.

D. *Cancel Flood Attack*

Exchanges that charge gas to cancel orders are not attractive to market makers and are unlikely to develop liquid markets. This is because approximately 98% of market maker's orders are cancelled[10]. This can somewhat be mitigated using expiration times in lieu of cancel functions. However, expiration times do not have sufficient resolution since they operate at the granularity of block confirm time. This means granularity at the level of seconds is not viable, let alone milliseconds. The expiration feature as a replacement for cancel is mostly ineffective since the need for rapid cancels is when the price moves very fast and cancelling a large number of orders over and over is uneconomical in terms of gas price and the latency makes reacting to the market and moving orders impractical. On illiquid markets typical of DEX, an attacker can manipulate the price up and down fast enough that all bots are forced to cancel and move orders repeatedly.

IV. SPEED vs SAFETY

All the issues and attacks on on-chain exchanges arise from the modeling with the assumption that a blockchain is a cheap resource with predictable and low variance characteristics. If on the contrary, we choose to model the blockchain as an expensive resource with high variance characteristics, it becomes clear how we should model a decentralized solution. Until newer blockchains develop and have sufficient history of deterministic rather than probabilistic finality and latencies comparable to ping round trips, a practical approach is to ***decentralize safety-critical*** functions such as account custody and ***centralize speed-critical*** functions such as order matching. We believe that custody of user funds and user identity are safety-critical and choose to decentralize only those aspects first.

V. NON-CUSTODIAL MODEL

There are two main kinds of attack on custodial models. The first is the *heist*, where a significant portion of the funds is taken at once[11]. The attacker does not expect to be able to repeat the attack and would generally not care if the theft is detected immediately. The second is *skimming*, where small amounts of funds are taken repeatedly over a longer duration. The attacker generally would like to go undetected since repeating the attack as many times as possible is key to a big payout. Our model aims to provably prevent heists and prevent or detect most types of skimming using fraud-proofs.

A. Preventing Heists

For heists to be possible, the possibility for one person to control all coins in custody must be possible. Segregated accounts that only enable the depositor to withdraw only their own balance eliminate this possibility, since the attacker would now need the keys of most users of the platform. Ensuring that the user's keys are never sent across the network is also a fundamental requirement for this to work. Therefore all authentication must be using public/private keys only. Password or other shared secrets should never be used.

B. Trading Settlements

Movement of assets between users as part of trade settlement requires that both users sign the amounts and price they are ready to exchange. Although the exchange performs the match, the custody contract transfers assets after verifying user signatures and integrity of all changes. Executions need to meet specifications set by both users according to the signatures from their private key for the smart contract to move assets among users.

C. Malicious User

Since the only avenue to extract assets is via a user account, a compromised exchange would have to set up a malicious user and collaborate with it to move all assets into that user by supplying data in a way to overcome the checks of the custody contract. Since all executions need signatures from users and executions without assets backing them would be rejected, the exchange would have to rearrange existing legitimate orders to favor the malicious user. These can only occur in small pieces and are skimming attacks.

VI. DEFINITIONS

A. Asset

Any underlying instrument that is suitable for trading. Typically Ether or Ethereum tokens.

B. Address

An ethereum address controlled by a private key in the user's possession. This address acts as the user identifier.

C. Segregated User Account

Segregated user accounts hold assets segregated by user. The key constraint is that assets can only be deposited from or withdrawn to the address that represents its owner.

D. Verified Asset Swap

An atomic transfer of assets between two segregated user accounts after verification of signatures and full compliance with transfer request parameters from both users.

E. Custodian

An Ethereum smart contract that manages segregated user accounts and performs verified asset swaps.

F. Proof-of-violation

A cryptographic fraud-proof that the system has behaved in an unexpected manner, typically leading to undeserved transfer of assets. These proofs can be submitted to the custodian, which can address the situation appropriately. Not all instances are actual fraud; some could result from programming errors such as rounding errors or extreme conditions such as 51% attacks.

G. Exchange

An exchange is a centralized entity that maintains an order book and generates verified asset swap requests, which it then submits to the custodian.

VII. PROTOCOL

We develop and prove the protocol using graph theory. In our model, entities that can hold assets are nodes and all possible asset movements are directed edges. We limit the edges that can exist on any node by defining a set of constraints that enumerate all possible edges in the graph.

A. Segregation Constraint

A user is the only one who can only deposit into and withdraw assets from his account. This constraint removes the ability of a compromised exchange to directly steal most or all of the funds in the exchange. This requires the attacker to create a malicious user account and move funds to it first.

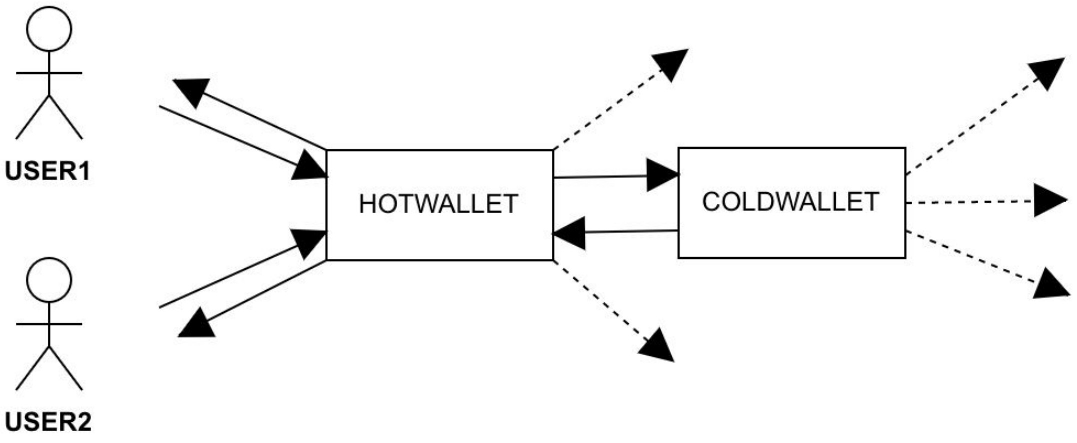

Fig. 1 The standard hot/cold wallet structure

The standard hot/cold wallet system has no constraints on how funds come in and go out and therefore is highly susceptible to heists.

Axiom 1: Segregated User Accounts are safe

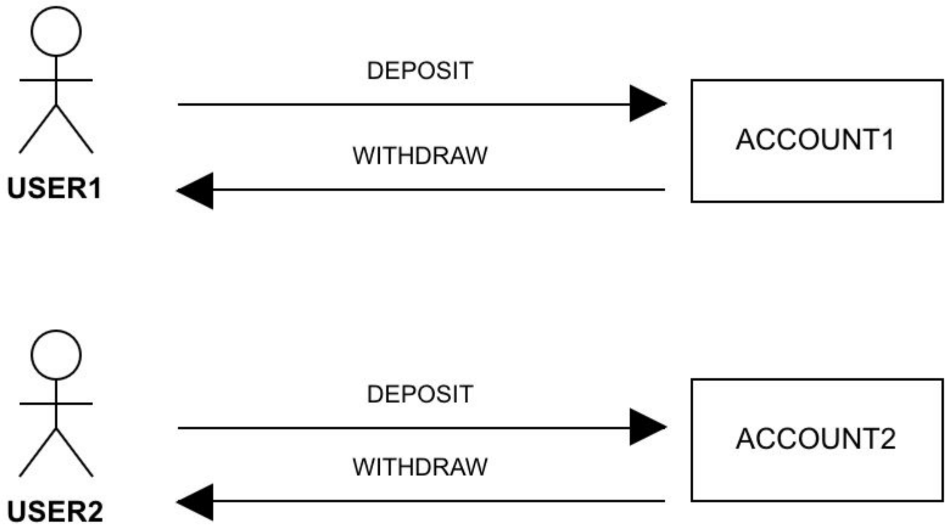

Fig 2. Segregated wallets

Segregated wallets allow deposit/withdrawal only by owner and are safe from heists since there is simply no path from the segregated wallet to anywhere else except their owner.

B. *Solvency Constraint*

Requests to transfer assets that exceed the user's balance will be rejected. Solvency constraint prevents a malicious exchange from creating orders without funds backing them, preventing siphoning attacks.

C. *Agency Constraint*

Any movements of funds to a different party cannot occur without verification of signature from the owner's private key. Asset swaps between users require signed instruction by both parties. To exchange 100 tokens for 1 ether, user1 would have to sign 'buy 100 tokens at limit price 1 ether' and user2 would have to sign 'buy 1 ether at limit price 100 tokens'.

D. *Integrity Constraint*

The total number of assets remains the same after a transfer and sum up to the prior amount. All transfers must satisfy the constraints (price and quantity) in the request.
The protocol constraints enumerated above ensure that all orders provably originate from users with sufficient funds to cover their positions.

Axiom 2: Verified asset swaps are safe

Assets that are swapped using a smart contract are safe from heists as long as the smart contract has no security defects.

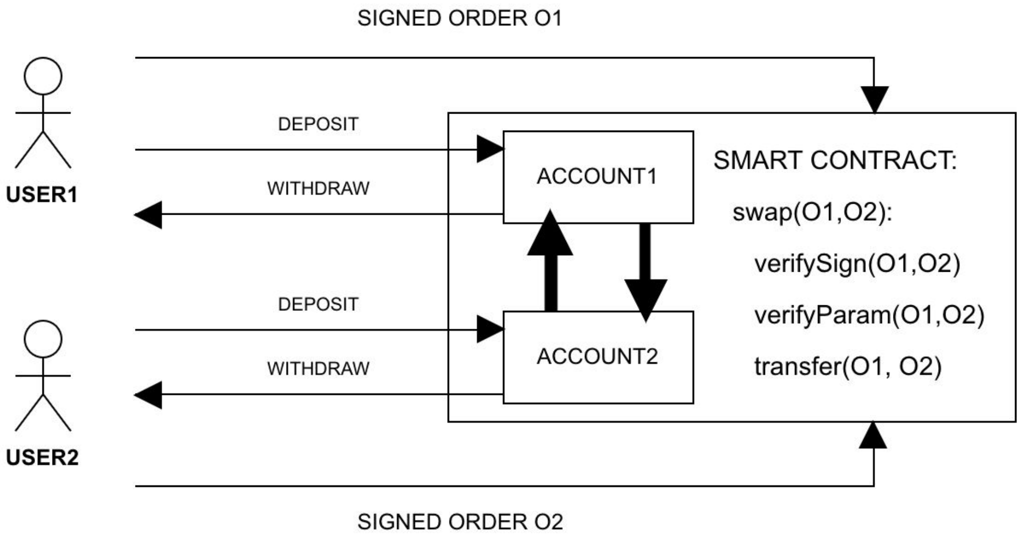

Fig 3. A smart contract swaps assets between segregated accounts

A smart contract can be used to safely swap assets between two segregated accounts. The transfer between user accounts is gated by the smart contract logic, which verifies user signatures on the order and that the order parameters such as price and quantity are a match. Exchanges that swap ERC20 tokens on-chain are good examples of this model[12]. As long as the custody and the settlement logic is on-chain, it's infeasible to steal coins from these accounts.

E. Delegated swaps using a centralized exchange

Instead of matching peer-to-peer, users can submit their orders to one or more matching entities (exchanges), which can do the matching and submit fills to the smart contract.

F. Protocol detail

1. User deposits into segregated account on Custodian
   a. User u_i deposits asset t_i into his segregated account A_i from ethereum address e_i
   b. Custodian updates user balance of t_i for A_i
2. Exchange updates balance of A_i after N confirmations enabling trading of new balance
3. User u_i submits signed order $\boldsymbol{\rho}_i$ with signature $\boldsymbol{\sigma}_i$
4. Exchange validates order $\boldsymbol{\rho}_i$ against signature $\boldsymbol{\sigma}_i$
5. Exchange countersigns order $\boldsymbol{\rho}_i$ with signature $\boldsymbol{\varsigma}_i$
6. Exchange updates order $\boldsymbol{\rho}_i$ in orderbook
7. Exchange matches orders $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ to creates execution $\mathcal{E}_i$
8. Exchange periodically submits execution set $\mathcal{E}$ to custodian
9. Custodian processes execution set $\mathcal{E}$
   a. Custodian verifies user signatures $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ and execution parameters (price, quantity, expiration, etc.) against orders $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ for each $\mathcal{E}_i$ in $\mathcal{E}$
   b. Custodian settles asset balance between users u_1 and u_2 of matched orders
   c. Custodian ensures integrity of all asset balances ensuring all assets add up
10. User u_i initiates a withdraw request of n t_i from his segregated Account A_i
11. Custodian moves n t_i to user's Ethereum Address e
12. Fees if any are moved from Custodian to a staking contract. There is never any assets flowing to Exchange

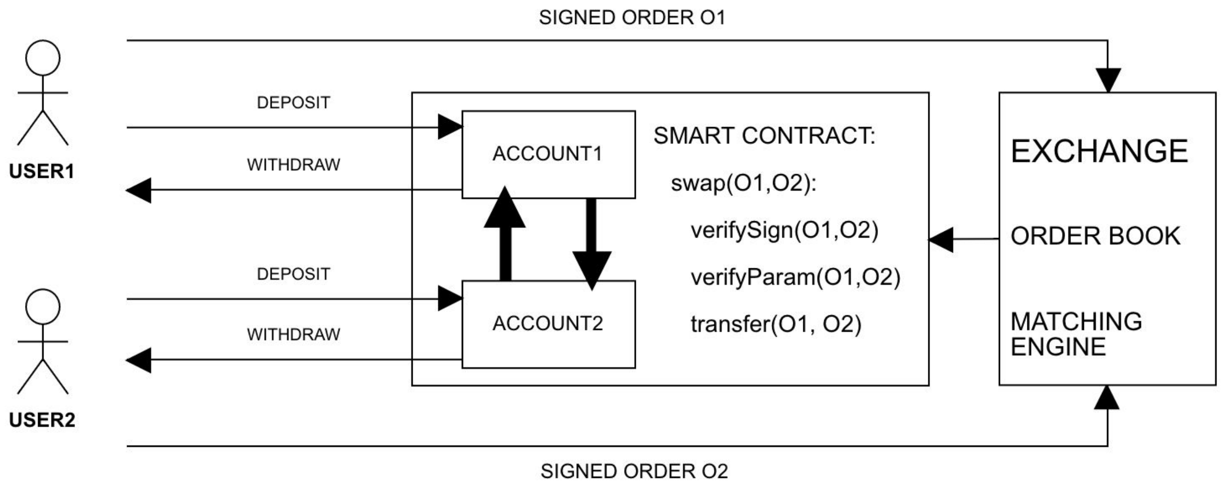

Fig 4. Off-chain matching and syncing to smart contract

Axiom 3: Delegated swaps are safe from heists

Protocol constraints prevent a compromised exchange from a large and swift heist and would be forced to play within the rules of the custodian. This means the attacker is likely to focus on the centralized component, the exchange. A compromised exchange can only manipulate its inputs (signed orders) and its outputs (executions). Since the only way to extract assets is as a user, an attacker would need to create conditions to give a colluding user marginal advantages in every trade. This requires waiting for victim users to place orders and then manipulate order books create fraudulent executions that look legitimate to the smart contract. Done long enough and often enough, this could be economically viable to perform. However, most of these attacks can be addressed using fraud-proofs.

VIII. PROTOCOL SECURITY

The exchange cannot synthesize or alter orders with the signature of a legitimate user and the protocol constraints prevent catastrophic theft. A compromised exchange however may collude with and favor a malicious user using a variety of skim attacks. To favor a colluding user, the exchange would need signed orders from victim users which would be filled at an adverse price. Since the exchange cannot synthesize or alter orders, it may try to manipulate legitimate orders to use them for victimization. Note that even a coarse expiry time on the orders greatly reduces the impact and detection burden of the attack. Note that this is not a comprehensive list of all possible attacks but a framework to quickly address them by quickly adding fraud-proofs when a new defect is discovered.

A. *Replay attack*

A simple way to synthesize a victim order is to simply remember a past order and use it when the price is adverse for the victim. The exchange simply duplicates an order and fills it at a price favoring a colluding user. Custodian contract prevents replay attacks by the tracking order ids of all matched orders and rejecting duplicates.

B. *Cancelled order attack*

A variant of the above is when the exchange simply fills an order that was cancelled. Storing cancelled order ids in the smart contract is impractical, since cancels are numerous and cancellation need to be fast. However, proof-of-cancellation can be used as a fraud-proof that can be submitted to the Custodian. The custodian can verify that a cancelled order has been filled and halt the exchange.

C. *Price-time priority violation attack*

Even non-cancelled orders can be used for attacks by simply withholding them from the orderbook and waiting for an adverse price movement and matching with a colluding user. Such violations can be detected by the user using price-time fraud proof: an execution exists for an order with a better price but later timestamp than the victim order.

D. *Non-acknowledgement attack*

An exchange may refuse to accept legitimate orders from a user due to operational policies such as rate limiting or network issues; these do not produce fraud-proofs. In contrast, a non-acknowledgement attack occurs when the exchange takes a user's signed order and instead of replying with its own signature, saves the submitted order for skimming. It then waits for the price to move and then matches it at an unfavorable price, siphoning profit to a colluding user. If this happens, a user can detect immediately when he sees one of his unacknowledged orders filled. The fraud proof for this is the same as the price-time fraud-proof.

E. *Suppressed Executions Attack*

While executions cannot be synthesized or duplicated, they can certainly be suppressed, enabling a compromised exchange to simply suppress losing trades of a colluding user from being synced to the smart contract and withdrawing funds while the smart contract is still in the stale state. This is a variant of TOCTOU attack[13]. Users who observe withdrawals being processed while the Custodian is in a stale state can submit a fraud proof preventing the attacker from withdrawing.

F. *Mitigation of skim attacks*

The exchange would need to sync executions to the smart contract containing a fraudulent execution before the assets can be withdrawn. The syncing will cause anyone monitoring the executions to detect the anomaly instantly and submit fraud-proofs as described below, preventing the attacker from stealing funds.

Axiom 4: There are 5 possible skimming attacks
A compromised exchange can rearrange orders that flow into and executions that flow out of the matching engine. There are four possible rearrangements: suppress, pass faithfully, duplicate and reorder (i.e., pass after a delay)
There are two possible order operations: Cancel and create. Updates are simply cancel and create.

Faithful passes are not attacks, so we have the following attack matrix:

Order	**Suppress**	**Duplicate**	**Reorder**
Cancel	Non-Acknowledgment	No effect	Cancel Order Attack
Create	Non-Acknowledgment	Replay	Price-time Priority

Execution	**Suppress**	**Duplicate**	**Reorder**
Match	Execution Suppression	Prevented	No effect

Table 1. Combinatorial attacks on leverj protocol

All these attacks are described above and have fraud-proofs that can be submitted to the Custodian contract to disincentivize skimming attacks.

IX. SUBMISSION OF FRAUD-PROOFS

A malicious actor would need to withdraw their funds after a fraudulent action. As long as there is a sufficient window of time to detect and submit fraud proofs, the smart contract can prevent the withdrawal or disable further action on the Custodian. Other users can detect the state change and withdraw their assets at their convenience. We propose a waiting period for every user after a withdrawal request, sufficient enough for any fraud proofs to be submitted. The exchange and smart contract would not accept any transactions for a user until their withdrawal is processed. This slight inconvenience is a justifiable tradeoff for the added safety. On the verification of a fraud proof, the exchange would halt. The perpetrator's balance could be awarded to the fraud-proof submitter, making compliance a form of mining.

X. CONCLUSION

Centralizing speed-critical functions and decentralizing safety-critical functions enable us to avoid on-chain exchange attacks vectors. The balance between high speed and on-chain security can be accomplished to a degree that enables us to add significant value today bringing the best of both worlds and provides an easy migration path to full decentralization on high-speed chains of the future.

XI. FUTURE WORK

1. For simplicity, this paper focuses only on spot-trading using limit orders. Adjustments for trading fees, margin trades and liquidations are out of scope.
2. In high-speed operational environment, coalescing executions in order to reduce the state load may be beneficial.
3. An incentive structure for submitting fraud-proofs may encourage decentralized monitoring and detection of exchange compromise. It may be as simple as awarding exchange fees collected or the account balance of the malicious user.
4. Current version assumes the Custodian will expire at a point in time when all assets are withdrawable by the user. This part could benefit from transparent UX.

REFERENCES

[1] https://bitcoin.org/bitcoin.pdf
[2] https://blog.ethereum.org/2016/05/09/on-settlement-finality/
[3] https://bitcoil.co.il/Doublespend.pdf
[4] https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/
[5] https://www.reddit.com/r/EtherDelta/comments/6wtbvu/etherdelta_performance_2017829/
[6] http://www.swende.se/blog/Frontrunning.html
[7] https://bitshares.org/
[8] https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/
[9] http://hackingdistributed.com/2017/08/13/cost-of-decent/
[10] https://www.sec.gov/marketstructure/research/highlight-2013-01.html
[11] https://en.wikipedia.org/wiki/Bitfinex_hack
[12] https://swap.tech/whitepaper/
[13] https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

APPENDIX—Solidity Code Samples

A. Segregated Accounts

```
 function() payable {
     ethers[msg.sender] = SafeMath.add(ethers[msg.sender],msg.value);
 }

 function depositToken(uint256 _value) returns (bool result){
     tokenCount = SafeMath.add(tokenCount,_value);
     tokens[msg.sender] = SafeMath.add(tokens[msg.sender],_value);
     return token.transferFrom(msg.sender, this, _value);
 }

function withdraw(address _user, uint256 _eth, uint256 _tokens)
          onlyOwner notDisabled {
     require(SafeMath.add(lastStateSyncBlocks[_user] ,
             withdrawBlocks) <= block.number);
     send(_user, _eth, _tokens);
 }
```

B. Cancelled order Replay fraud-proof disables smart contract

```
function notifyReplay(uint[] orderUINT, bool isBuy, address user,
                        uint8 v, bytes32 r, bytes32 s){
     Order memory order = Order(orderUINT[0], orderUINT[1], orderUINT[2],
                         orderUINT[3], orderUINT[4], isBuy, user);
     require(isVerified(order, owner, v, r, s));
     require(SafeMath.sub(order.qty, order.cancelled) < filled[order.uuid]);
     disabled = true;
 }
```

C. Users can recover balance from disabled contract

```
 modifier isDisabled{
     require(disabled);
     _;
 }

function recoverFunds() isDisabled {
     send(msg.sender, ethers[msg.sender], tokens[msg.sender]);
 }
```