Real-Time Cryptocurrency Exchange without Trusted Clocks

Greg Capra- greg_capra@berkeley.edu, EECS MEng
Ying-Yi Liao- yingyi_liao@berkeley.edu, EECS MEng

Section 1: Problem Definition and Motivation

Cryptocurrency exchanges are services where traders can buy, sell or exchange cryptocurrencies for other digital or fiat currencies. Since automated programs are usually used for high frequency trading and arbitrage, traders are able to modify their trading positions within milliseconds to respond to the rapid price fluctuations. This requires exchange services to be responsive in real-time. Unfortunately, most existing real-time cryptocurrency exchanges rely on third parties (TPs), which puts traders' funds at risk of being stolen or hacked [1,6]. Tesseract [1] remedies this risk by using Intel SGX enclaves [2] to ensure that the traders' funds can never be stolen. A trusted clock is used within this protocol to ensure that the attacker cannot feed the SGX enclave with self-mined blocks and execute an eclipse attack. However, the assumption of a trusted clock does not always hold. In this work, we propose an extension to Tesseract that thwarts the need for trusted clocks while maintaining the real-time feature and preserving security. This approach, which utilizes bidirectional off-chain channels together with timelocked multisignature contracts, seeks to achieve these goals without introducing additional attack vectors.

Eclipse attacks are attacks where an adversary sends fake blockchain data to an enclave in order to steal money by trading fake currency for real currency. During an attack an adversary can only send blocks at least twice as slow as it normally would (since we're assuming it has less than ½ the computing power of the blockchain network). Thus, a trusted clock can detect that blocks are being received slower and make some sort of correction in form of a rule- for example, to wait a certain amount of additional block confirmations before publishing. So if a trusted clock is not an assumption that holds in all cases, the Tesseract exchange will have an open attack vector for malicious nodes to exploit. Not only might this attract bad actors, but also reduce confidence in the exchange itself as money will be stolen.

Section 2: Related work and comparison

So far, there are no solutions for this issue. However, there are a number of papers on both TEEs and off-chain channels that will be referenced for this work. The existing system Tesseract is a secure, real-time cryptocurrency exchange which achieves both security and performance by using Intel SGX enclaves as trusted execution environments. They assume SGX has access to a trusted clock in order to prevent eclipse attacks, however, this assumption does not always hold, so we want to improve this system into the one that is still free from such an attack but without the use of trusted clock.

There is another kind of exchange that integrates off-chain bidirectional payment channels with TTPs [1,6]. The advantage of this type of exchange is that it allows traders to do a high frequency of small-amount payments without paying relatively large amounts of transaction fees because they are transferring off-chain. The disadvantage of this structure is that the TTP can always steal the most recent amount that was funneled through it, so it makes the traders vulnerable to theft for large trades.

Centralized exchanges, exchanges with off-chain channels or TTPs, and Tesseract all have their own pros and cons. What we want to do is to combine these to keep the main advantages- efficient, real-time, and secure.

In our approach, we keep the cryptocurrency swapping process off-chain in bi-directional channels, therefore the trading efficiency can be much better than other on-chain, decentralized exchanges. Our only on-chain interactions are when depositing (initiating an account with Tesseract), or withdrawing funds (leaving Tesseract). Additionally, we utilize the benefits of TEEs, multisignature contracts, and other cryptographic logic to guarantee the security of users' assets which centralized exchanges are unable to reliably do. Thus, our exchange paradigm is a great combination of the advantages from both without the potholes.

Section 3: Approach

Our design combines TEEs with off-chain channels so that TEEs are no longer required to interact with the blockchain directly. In order words, the TEE running on a Tesseract server doesn't interact with blocks fed from any chain. All work is done offline between users' clients and the Tesseract server, which is later validated by contracts before doing on-chain withdraws. Thus, we do not need to rely on the trusted clocks to determine the rate of blocks (or fake blocks) reaching the server.

The major challenge for this approach is to specify a way that TEEs can interact with off-chain channels (in order to remove reliance on a clock) while preserving security against an Eclipse attack, the main purpose of using the trusted clock. We also need to make sure that additional attack vectors aren't introduced by utilizing such an approach, and that latency/throughput is not compromised. However, given time constraints, we will leave the analysis of the latter to future work.

Establishing off-chain channels between each user and the server is done using timelocked multisignature contracts. The purpose of a timelock is to allow a user that doesn't perform any actions with his/her money to withdraw after a certain time limit. In centralized exchanges you're assuming that the exchange will reliably hold your funds. However, in our system everything is held in shared contracts, which are validated by the blockchain ledgers. The benefit of using multisignature contracts is that they help prevent against collusion. If Alice is trading bitcoin to Bob, she must sign off on the transaction in addition to Tesseract. So if she detects that either the Tesseract server (not the TEE but the server it runs on), Bob, or both are malicious she will refuse to sign the transaction and wait until after the timelock to withdraw her money.

The service is broken into three phases- Epoch, Claim, and Withdraw, in that order. In the Epoch phase users can deposit into the appropriate multisignature contracts recognized by Tesseract and conduct trading by sending requests to the Tesseract server. During the Claim phase, users submit claims to the contracts containing cryptographic justification that they deserve certain funds due to offline trading. These consist of the necessary signatures (for Bob claiming Alice's funds, he would need both Alice's and Tesseract's) as well as the unique IDs of the coins the user is claiming. The contracts handle the logic to determine if the claimers deserve said funds. Then, in the withdraw phase, once any user submits a request to withdraw it will withdraw all funds for all users based on the claims submitted and validated in the prior phase. Any unclaimed money will get returned to the original owners, so no money will be lost. Of course, these phases are mutually exclusive. In other words, users can only deposit and trade during the

Epoch phase, only submit claims during the Claim phase, and only withdraw during the Withdraw phase. After the Withdraw phase ends, the next Epoch phase immediately starts. If users wish to keep trading, they can deposit funds and start over again.

Our protocol is listed below, split into client (Tesseract users) and server (Tesseract) parts. Note that the “Order” sub-protocol is the Tesseract user’s responsibility if attempting to trade during the Epoch phase, while the “Trade” sub-protocol is Tesseract’s corresponding responsibility. Also note that only users need to deposit, withdraw, and submit claims- that functionality is abstracted away from Tesseract in our protocol so the server doesn’t have to interact with on-chain events and thus depend on data from the blockchain. We have written the pseudo code in terms of Alice and Bob who are swapping BTC (Bitcoin) for ETH (Ethereum) respectively. In the following passage after the protocol we walk through an in-depth example to demonstrate how it works in practice, and provide a visual representation.

1 Client Protocol

Goal: A secure, real-time, decentralized cryptocurrency exchange protocol without the need for trusted clocks. This builds off the Tesseract design.

Protocol 1 Deposit

Notes/Assumptions: We specify a user Alice A who owns BTC, a user Bob B who owns ETH, and Tesseract server S. Denote the global, Tesseract-Ethereum smart contract by $Contract_{ETH}$ and a timelocked, 3 of 3 multisignature Bitcoin contract created by Alice as $M_{A,B,S}$. The public key notation $PK_{A,BTC}$ indicates Alice's Bitcoin address. The tuple $(Deposit_A, splits_A)$ indicates 2 arguments of the deposit operation: the deposit amount and number of coin splits (security parameter, explained later). WLOG, suppose Alice wants to deposit 10 Bitcoin $(Deposit_A)$ and Bob wants to deposit 100 Ethereum $(Deposit_B)$.

Initial State: $PK_{A,BTC}$ has $Deposit_A$ and $PK_{B,ETH}$ has $Deposit_B$

Goal: $PK_{A,BTC}(Deposit_A) \rightarrow M_{A,B,S}$ and $PK_{B,ETH}(Deposit_B) \rightarrow Contract_{ETH}$

The protocol:

1. if (currentPhase $\neq$ Epoch) $\rightarrow$ quit
2. $(Deposit_A, splits_A)$: $PK_{A,BTC} \overset{UTXO}{\rightarrow} M_{A,B,S}$, $(Deposit_B, splits_B)$: $PK_{B,ETH} \rightarrow Contract_{ETH}$
3. (a) If $Deposit_A$ and $Deposit_B$ successful, secure channels between A and S and B and S are established via contracts $M_{A,B,S}$ and $Contract_{ETH}$ respectively. Trades can now happen.
   (b) If $Deposit_A$, $Deposit_B$ fail, A and B need to rerun **Deposit**.

Protocol 2 Order

Initial State: $M_{A,B,S}$ has $Deposit_A$ and $Contract_{ETH}$ has $Deposit_B$

Goal: $Order_A(BTC_{Sell}, ETH_{Buy}, Rate) \rightarrow S$ and $Order_B(BTC_{Buy}, ETH_{Sell}, Rate) \rightarrow S$

The protocol:

1. if (currentPhase $\neq$ Epoch) $\rightarrow$ quit
2. $(Order_A, UserID_A)$: $A \rightarrow S$, $(Order_B, UserID_B)$: $B \rightarrow S$

Protocol 3 Claim

Initial State: A has $Address_B, Signature_B, Signature_S, Message_{Coin_{B,ETH}}(coinIndex, coinID, nonce), Hash($
B has $Address_A, Signature_A, Signature_S, Message_{Coin_{A,BTC}}(nonce), Hash(Message_{Coin_{A,BTC}})$

Goal: $Coin_{B,ETH} \rightarrow A_{Contract_{ETH}}$ and B: $Sign(Message_{Coin_{A,BTC}})$

The protocol:

1. if (currentPhase $\neq$ Claim) $\rightarrow$ quit
2. $A : Address_B, Signature_B, Signature_S, Message_{Coin_{B,ETH}}, Hash(Message_{Coin_{B,ETH}}) \overset{claim}{\rightarrow} Contract_{ETH}$,
   $B : Sign(Message_{Coin_{A,BTC}})$
3. if(verifySigner($Address_B, Signature_B, Message_{Coin_{B,ETH}}, Hash(Message_{Coin_{B,ETH}})$) == True && verifySigner($Address_S, Signature_S, Message_{Coin_{B,ETH}}, Hash(Message_{Coin_{B,ETH}})$) == True && verifyCoinID($Address_B, Message_{Coin_{B,ETH}}(CoinIndex, CoinID, Nonce)$ == True && $Nonce_{received} > Nonce_{previous}$):
   - $Coin_{B,ETH} \rightarrow A_{Contract_{ETH}}$

Protocol 4 Withdraw

Initial State: $M_{A,B,S}$ has $Coin_{A,BTC}$ and $A_{Contract_{ETH}}$ has $Coin_{B,ETH}$

Goal: $A_{Contract_{ETH}} \overset{Coin_{B,ETH}}{\rightarrow} PK_{A,ETH}$ and $M_{A,B,S} \overset{Coin_{A,BTC}}{\rightarrow} PK_{B,BTC}$

The protocol:

1. if (currentPhase $\neq$ Withdraw) $\rightarrow$ quit
2. any user: withdraw() $\rightarrow Contract_{ETH}$ then $A_{Contract_{ETH}} \overset{Coin_{B,ETH}}{\rightarrow} PK_{A,ETH}$
3. B: $Signature_B, Signature_A, Signature_S, Message_{Coin_{A,BTC}}(nonce), Hash(Message_{Coin_{A,BTC}}) \overset{redeem}{\rightarrow} M_{A,B,S}$ then $M_{A,B,S} \overset{Coin_{A,BTC}}{\rightarrow} PK_{B,BTC}$

2 Server Protocol

Protocol 5 Trade

Notes/Assumptions: Similar to the Client protocols we inherit the notation A, B, S, $M_{A,B,S}$ and $Contract_{ETH}$. The public key notation $PK_{A,ETH}$ indicates Alice's Ethereum public key. The messages m_1 and m_2 indicate the match details provided to the two parties by S once a match is finalized via the internal order book. WLOG, suppose Alice is interested in swapping 1 Bitcoin (TX_A) for 10 Ethereum (TX_B), which Bob has. This protocol would occur after and/or during protocols 1 and 2 on the client, so we are assuming S has already matched A and B. Below, the message details are m_1 and $m2$, which contain $Address_A, Address_{M_{A,B,S}}, nonce$, and $Address_B, Message_{Coin_{B,ETH}}(CoinIndex, CoinID, Nonce), Hash(Message_{Coin_{B,ETH}})$ respectively.

Initial State: $M_{A,B,S}$ has $Deposit_A$ and $Contract_{ETH}$ has $Deposit_B$. If no trades occur, both will be refunded after the end of the Withdraw phase.

Goal: $S : Address_B, Signature_B, Signature_S, Message_{Coin_{B,ETH}}, Hash(Message_{Coin_{B,ETH}}) \rightarrow A$, $S : Address_A, Signature_A, Signature_S, Message_{Coin_{A,BTC}}, Hash(Message_{Coin_{A,BTC}}) \rightarrow B$

The protocol:

```
1. if $(numSplits_A \neq numSplits_B) \rightarrow$ quit
2. while((btcTraded $< 1BTC$) && (ethTraded $< 10ETH$)):
    • $A_{sk}((TX_{A^*}, m_1) \rightarrow B) \rightarrow S$, where $TX_{A^*} \leq 1BTC$
    • $S_{sk}A_{sk}((TX_{A^*}, m_1) \rightarrow B) \rightarrow B$. $B$ now has full control of funds in $M_{A,B,S}$
    • $B_{sk}((TX_{B^*}, m_2) \rightarrow A) \rightarrow S$, where $TX_{B^*} \leq 10ETH$
    • $S_{sk}B_{sk}((TX_{B^*}, m_2) \rightarrow A) \rightarrow A$. $A$ now has full control of funds in $B_{Contract_{ETH}}$
    • if((verify($TX_{A^*}$) == success) && (verify($TX_{B^*}$) == success)):
        – $S$.updateAccounts($A$,$B$)
        – btcTraded += $TX_{A^*}$, ethTraded += $TX_{B^*}$
    • else:
        – refund($TX_{A^*} -> A$) && refund($TX_{B^*} -> B$)
3. WLOG, if($A$ : **trade**($TX_{B^*} \leq 10ETH$) $\rightarrow C$):
    • repeat above
    • for($TX_{B^*}$ in $Contract_{ETH}$):
        – $m_2$: $nonce += 1$
        – $S$: issueToken($S_{sk}B_{sk}((TX_{B^*}, m_2) \rightarrow A)$) $\rightarrow C$
```

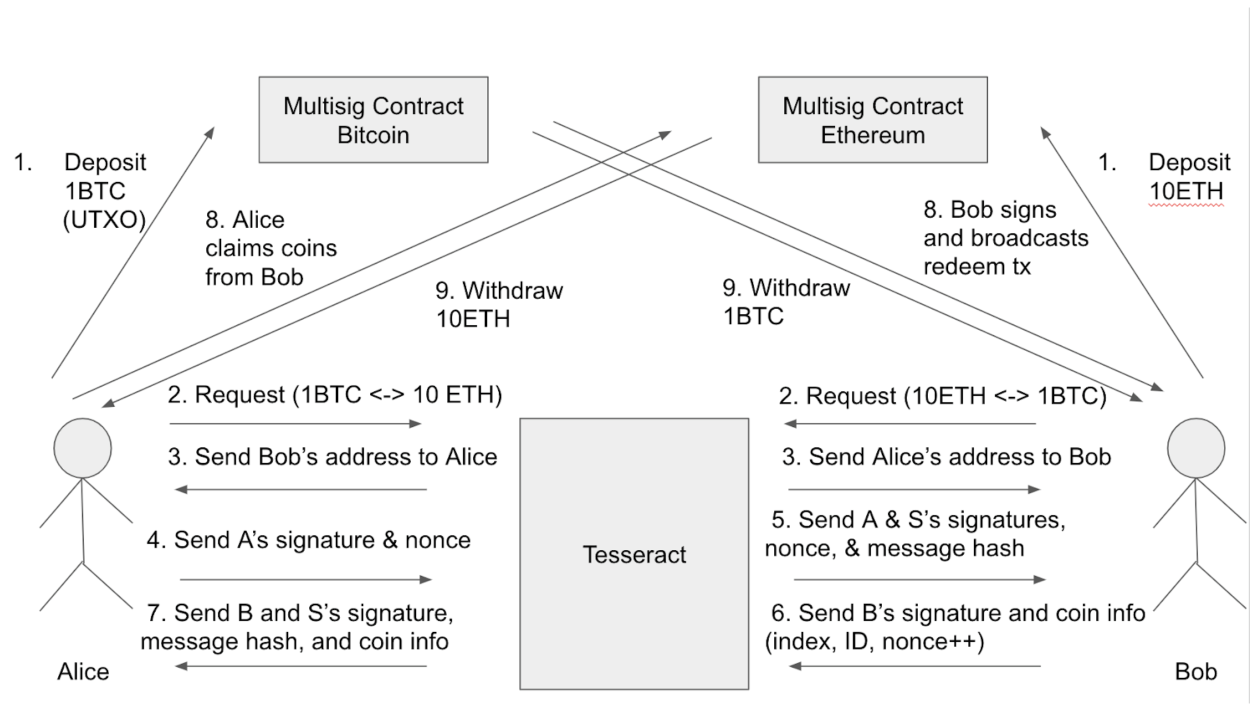
Let's suppose Alice wants to exchange 1 BTC for 10 ETH, and Bob wants to exchange 10 ETH for 1 BTC. S denotes the Tesseract server which houses (or can simply send requests to) the TEE. In the Deposit operation (during the Epoch phase), Alice should create a timelocked, 3 of 3 multisignature contract shared between herself, S, and Bob, and deposit a certain amount of Bitcoin which is equal to or larger than the amount she wishes to trade during the Epoch. In reality, Alice would do this by broadcasting a UTXO spend transaction to the multisig address after creating it. Bob, who holds ETH, can simply deposit his funds during the Epoch into the Tesseract smart contract which has the functionality to operate in a timelocked, multisignature manner (Ethereum implementation is simpler due to the Turing-complete nature of writing contracts in Solidity). Both Alice and Bob's operations thus far are on-chain and can therefore be validated. If they see that for some reason their deposit requests were not accepted by the respective blockchain, they will need to try again or wait until the next Epoch.

Next, during the Trade operation, Alice and Bob both send order requests offline to S. It is still the Epoch phase, so S is conducting the appropriate order book matching. We will assume Alice and Bob submitted equivalent trade requests and were matched by S. S then sends a message to each indicating the match details, including each other's cryptographic addresses. Now they can engage in an offline swap. To do so, Alice signs a new transaction sending 1 BTC from the timelocked, multisignature contract created in the deposit phase to Bob, and sends to S. In this transaction should be a clause that prevents Bob from redeeming until the Withdraw phase. S also signs this transaction and sends to Bob. Because both parties of the the multisignature contract have already signed the transaction, Bob can now sign and broadcast a redeem transaction from the multisig after the timelock (so during the Withdraw phase). Then Bob will generate his own signature, signing the coins he deposited in the Tesseract smart contract over to Alice, and send this to S. There is no need for him to specify a redeem timelock clause here since the Ethereum smart contract will take care of this logic. S completes this by generating its own signature and sending to Alice. Alice now has full control of Bob's funds. This preliminary approach leaves out some security considerations that we will cover later in section 5.

Bob can immediately redeem his transaction after the Withdraw phase starts, so the Claim phase doesn't apply to him (the phases are synced up between ETH and BTC, but Claim only applies to ETH). However, this phase is arguably the most important for those receiving ETH since they will need to submit claims to the smart contract via the Claim operation. This operation will require Alice to submit the signatures from Bob and S, Bob's Ethereum address, and the coin information. The smart contract will take care of all validation, making sure the transaction was signed by the right people and on the right data. If it passes, the funds will be made available for Alice to redeem during the Withdraw phase. If it fails, Alice will be unable to withdraw and will need to submit additional claims before the end of the phase.

The Withdraw operation is quite simple- for ETH only one user needs to call the Withdraw function in the smart contract for it to invoke a process that withdraws all funds to the correct users based on the claims made in the previous phase. If valid claims were not placed on a user's funds, the funds are automatically redeemed from the smart contract to their Ethereum address. For BTC, a user will need to submit a redeem transaction during the Withdraw phase otherwise the money can be redeemed by the original owner (remember it was originally timelocked).

Our full code implementation is laid out in the next section. In section 5 we dive into some more advanced scenarios such as continual trading. We also analyze the security of our protocol and how it holds up to cases when parties are malicious.


Multisig Contract Bitcoin
Multisig Contract Ethereum
1. Deposit 1BTC (UTXO)
1. Deposit 10ETH
8. Alice claims coins from Bob
8. Bob signs and broadcasts redeem tx
9. Withdraw 10ETH
9. Withdraw 1BTC
2. Request (1BTC <-> 10 ETH)
2. Request (10ETH <-> 1BTC)
3. Send Bob's address to Alice
3. Send Alice's address to Bob
4. Send A's signature & nonce
5. Send A & S's signatures, nonce, & message hash
6. Send B's signature and coin info (index, ID, nonce++)
7. Send B and S's signature, message hash, and coin info
Tesseract
Alice
Bob

<u>Section 4: Implementation</u>

a. Tesseract Server

Our Tesseract server will be executed in the enclave of Intel SGX and it only supports Rust as its programming language, so we chose Rust as our programming language to implement the Tesseract server.

<u>Data Structures</u>

There are two self-defined structs in the server, `RequestInfo` and `OrderBook`. `RequestInfo` is used for storing the JSON request sent from the client. The first field of `RequestInfo` is `request_type`. It is a string that has two possible values, “order” or “trade”. If the `request_type` of `RequestInfo` is “order”, it means that this request is an order request sent from the client in order to exchange BTC for ETH or to exchange ETH for BTC by a certain rate. An order request consists of six fields:

- `user_id` indicates the client's user ID.
- `rate` indicates the ETH/BTC rate the client wants to trade for.
- `buy_crypto_type` and `sell_crypto_type` indicate the client wants to exchange BTC for ETH or to exchange ETH for BTC.
- `buy_amount` and `sell_amount` indicate the amount of cryptocurrencies the client wants to trade.

For example, the request {"request_type": "order", "address": "","coin_index": 0,"coin_id": 0,"nonce": 0, "user_id": "alice12345", "rate": 10, "buy_amount": 1, "sell_amount": 10, "buy_crypto_type": "BTC", "sell_crypto_type": "ETH"} means that it is an order request sent from user ID alice12345, and she wants to trade 10 ETH for 1 BTC at the ETH/BTC rate = 10. There are four fields with default values (empty string for string type and 0 for integer type), `address`, `coin_index`, `coin_id`, `nonce`. They are only used in the Trade request, so are set to default values in the order request.

If the `request_type` of `RequestInfo` is “trade”, it means that this request is a trade request sent from the client in order to send the information of token (a trade unit which has small value, referred as coin afterwards) that this client wants to transfer to the other trade party. A trade request consists of four fields:

- `address` indicates the the other trade party’s ETH address or BTC public key.
- `coin_index` indicates which index the coin is stored at the client’s `balance` array in the multisigature contract (only applicable to ETH).
- `coin_id` indicates the coin’s global ID which is unique in that kind of cryptocurrency.
- `nonce` indicates the timestamp of this coin. It will be incremented whenever this coin’s owner changes.

Among all the coins with same global ID, only the one with the largest (which means latest) nonce is valid, so the global ID of coin is still unique. By this mechanism, we can ensure that only the latest owner has the right to claim this coin. For example, the request {"request_type": "trade", "address": "0xF21425bbE5e556A490fB2d11774d1788e944B3d3","coin_index": 0,"coin_id": 0,"nonce": 1, "user_id": "", "rate": 0, "buy_amount": 0, "sell_amount": 0, "buy_crypto_type": "", "sell_crypto_type": ""} means that it is a trade request that the client wants to transfer her coin to the other trade party whose address is "0xF21425bbE5e556A490fB2d11774d1788e944B3d3". The coin she is transferring stores in the first index (coin_index 0) of her `balance` array in the ETH smart contract, its global ID happens to be 0 and its nonce will be incremented automatically from 0 to 1 so the other trade party will get this coin with latest nonce.

`OrderBook` is used for storing each order request sent from clients. Thus, two `OrderBook` objects will be instantiated for the server, one stores the Sell-ETH/Buy-BTC orders and the other stores the Buy-ETH/Sell-BTC orders. The underlying data structure of `OrderBook` is a priority queue. In the Sell-ETH/Buy-BTC order book, the order with the lowest ETH/BTC rate will go to the top of the priority queue, while in the Buy-ETH/Sell-BTC order book, the order with the highest ETH/BTC rate will go to the top of the priority queue. If the top orders of both order books have the same rate, then those orders will be matched and the users who send those order requests will be informed to start trading coins. After the trading process is finished, both orders will be popped out from the priority queues.

Request Handling

First, `Server` object of the `sync` crate is created and is binded to the ip address and port we use. Then, we build a socket using the function `use_protocol("rust-websocket")` of the `websocket` crate. After the socket is built, a new thread will be spawned for each new client connection. After creating the thread, the program execution will stay in a loop waiting for each request sent from the client. Every time when the server receives a new request, the code in the loop will be executed once. For each request, the JSON message sent from the client will be parsed into a `RequestInfo` object using `serde` crate (crate is how to call a library in Rust). Next, we use the `request_type` field to identify that it is an order request or a trade request. If it is an order request, then we push it into the order book (use the sell_crypto_type and buy_crypto_type to decide which order book it should be pushed into). Then, we immediately peek the top order of both order books to see if there is a match. If it is a trade request, then we call the `sign_message` function to start the signing process.

Signing Process

In the signing process, first we create a message to sign by concatenating `address`, `coin_id` and `nonce` into a single string. We don't need to include `coin_index` in our message to sign because it simply aids the multisignature contract in retrieving the coin from the `balance` array faster and nothing to do with verification (it's not globally unique). Next, we use `Sha256` object of the `crypto` crate in Rust to hash the concatenated string. Then, we use `hex` crate to decode the secret key of Tesseract and input the decoded secret key and the hash into the `sign` function of the `secp256k1` crate. This function uses the same signing algorithm (ECDSA: Secp256k1) as Bitcoin and Ethereum does, so the signature it generates can be verified in the Solidity code and Bitcoin Script. This `sign` function returns a `Signature` object and a `RecoveryId` object. There are two fields, `r` and `s`, in the former, and the value of latter is either 0 or 1. The inputs of `ecrecover` function, which is the function to verify if a signature is signed by a certain address in Solidity, are hash, v, r, s. We obtain hash from `Sha256`, r and s from the `Signature` returned by `sign` . Value v in solidity is either 27 or 28, which is exactly when the `RecoveryId` is 0 or 1, respectively. Thus, after the signing process, the receiver of the coin has the signature from the owning party, the signature from Tesseract, the hash of the signed message, the address of the owning party, and all the raw message information he needs (`coin_index`, `coin_id`, `nonce`) to claim the coin.

b. User's Client

The user's client can be written in any kind of language, but we chose Rust as well in order to communicate with the Tesseract server more easily, and also can reuse the `sign_message` function written in server. The implementation of client is more simple than server. First, it makes a connection to the server. Then, it enters a loop. In the loop, it first lets users to input in the terminal. The format we we input will be a JSON string with contains all the fields in `RequestInfo`. And then, it does the same thing as the server does. The request inputted by the user will be parsed into a `RequestInfo` object using `serde` crate, and also using the `request_type` field to identify that it is an order request or a trade request. If it is an order request, then the client will simply pack the JSON string into `Message` object of `websocket` crate and send it out to the server. On the other hand, if it is a trade request, then the client will ask the user to input his secret key in order to sign the message before sending tit out to the server. The original owner of the coin should sign it first before sending it to Tesseract, because the claimer of that coin will require both the signature of the original user and Tesseract. The signing process is the same as in the server. In reality, all of this would take place via a user interface where this functionality would be abstracted away and the user could just click on buttons to deposit, send trade requests, claim, and withdraw.

c. Multisignature Contracts- ETH and BTC

The logic for validating signatures, verifying phases, and transferring coin ownership is done through the multisignature contracts. For ETH this is simple- there is one, master Tesseract smart contract that contains a balance mapping for each user. The key is their public address, and the value is a struct called 'TwoWayChannel'. This struct has the monetary value deposited in the contract by the user, the number of splits (a safety parameter to be explained in the next section), and an array of the most recent claim requests for each coin (their money is split into individual coins for safety, again explained in the next section). As users submit claim requests to the contract these arrays are updated to only house the most recent valid claim request for each coin, or in other words, the valid claim request with the highest nonce. During the Withdraw phase the contract loops through these arrays and withdraws the coins to the claimers with that highest nonce. In order to keep state, any time a function is called in the contract another function called 'determineState' is run. This works by checking the current block number and comparing it to the block number that the most recent Epoch started. The sizes of each phase- Epoch, Claim, and Withdraw- are given at contract creation, so the contract is always able to calculate what phase it is currently in. This returns an Enum which it is able to compare with the called function to see if the operation should be permitted.

Implementing BTC, on the other hand, is not so simple. Currently there is no way to make a global smart contract that handles this logic for all users since Bitcoin Script is not turing-complete and therefore doesn't support this functionality. Instead, the logic must be done via individual 3 of 3 multisignature contracts- one for each trade. To handle state, each must be timelocked so a user cannot redeem until after what would be the start of the Withdraw phase for ETH. In addition, it must only be 3 of 3 if redeeming before the end of the Withdraw phase. If no redeem is submitted and the Withdraw phase has passed, the multisignature contract must have the flexibility to allow the original owner to submit a redeem transaction without the need for another signature (essentially 1 of 3 with the 1 being the original owner). This functionality is possible but must be explicitly written in raw Bitcoin script and compiled down to binary, and then included in a P2SH (Pay-to-Script-Hash) spend transaction. This essentially acts as sending a UTXO to the newly created multisig contract. If more time, we would like to write an API to handle this for us so that when a user's client initiates a Bitcoin deposit it would automatically create the compiled script with the right owners, access, and timelock logic, and then broadcast the spend transaction to it.

Section 5: Evaluation / Results

a. Security Analysis

Note that if S chooses not to sign one or both of the multisignature transactions in the trade operation, the money will revert to the original owners after the time limits. In the worst case where a malicious S does this consistently or goes offline, this simply amounts to a DOS attack. One interesting situation to consider is the case where one of the two parties (WLOG, let's say Bob) is malicious. In this case Bob might not choose to sign his 10ETH transaction over to Alice after Alice has signed her portion over to him. Regardless of whether or not S signs a spend transaction from the multisignature shared by S and Bob, Bob can choose not to sign it. After the time limit, the money would revert to Bob and he would effectively steal the 1BTC and retain his 10ETH, since his refund transaction will now be valid. To prevent against this, Alice, S, and Bob should engage in a transaction signing process where microswaps are facilitated until the full amount is reached or one party aborts. For example, Alice would start the swap by signing over .0001 BTC to Bob. If Bob never responds by signing over control of the corresponding amount of ETH to Alice, she can assume that Bob is malicious and refuse to continue with the swap. Thus, Alice will only lose a small amount. Since these transaction swaps are off-chain and depend only on the amount of time it takes Alice, Bob, and S to sign, doing many of these microswaps should be reasonably quick. One can easily see that an identical approach will help a party Alice protect against the situation where both Bob and S are corrupt. Both our protocol and code implementation utilize these microswaps as we believe this feature to be critical in building a secure service.

To do this we take in a parameter called 'numSplits' at request time- essentially the number of swaps the user would like to do to feel safe about transferring the given value (for example, 100 swaps on 1 ETH might be safe but not necessarily 100 swaps on 1000 ETH). For ETH we split the users deposit value into the number of specified coins and assign each a globally unique ID (used during claim validation). For BTC we simply sign incrementally larger and larger spend transactions as specified in [5] (for a value of 1BTC with 10 splits, the user would sign a tx that sends .9 BTC to himself and .1 to other party, then .8 to himself and .2 to other party, etc.). Note that the receiving party is always incentivized to broadcast the most recent transaction as that's the one where they receive the most money.

Next, consider the situation where Alice and Bob try to claim coins from a previous Epoch (not the one right before the current claim phase). Bob will be unable to claim these coins, since the global coin IDs will be different in the new cycle (these are determined in the current cycle at deposit time). Alice may not even have money deposited in the contract during this cycle. On the other hand, Alice will likely not be able to redeem funds from a past cycle's Bitcoin multisig either. This is because her redeem transaction is only valid during that cycle's Withdraw phase- as that was the logic built in. After this point Bob would have sent a redeem transaction to this multisig if Alice didn't. Therefore we can see that in either case users are incentivized to claim and redeem funds during the current cycle, and any funds that fail to be traded can easily be redeemed to the original users.

b. Ethereum Smart Contract Gas Consumption

______The main functions in the Ethereum smart contract are 'deposit', 'claim', 'withdraw', and 'determineState'. Currently, deploying the smart contract on a test network takes about 1,060,000 units of gas. Since gas is currently 20 Gwei at the time of writing, that means deploying the contract is about 0.0212 Ether, or $2.32 at the current price ($109/ETH). However, this only needs to be done once to go live. The contract is self sustaining and clears the values after each full cycle. However, setting new global parameters such as changing Tesseract's address or adjusting the phase lengths, will require redeployment. The 'determineState' function is very lightweight and only requires ~2,312 units of gas, or $.005. 'Withdraw' is also lightweight, requiring ~2,425 units of gas (also about $.005). 'Deposit' is relatively expensive, taking ~670,000 units of gas, or $1.49, for 1 ETH with 10 splits. For a deposit transaction of 10 ETH with 100 splits the transaction fee was .23 ETH, or $25. A $25 transaction fee for a $1,000 deposit is a bit high, and additional work should be done to improve the efficiency of this function so its gas consumption (and therefore transaction cost) can be reduced. Submitting a claim for an individual coin, regardless of value, is around 90,000 units of gas, or $.20. This function should also be optimized since users will have to pay reasonably high fees for claiming each coin in a microswap process.

c. Challenges/Future Work

One current limitation in this approach is continual swapping. Say, for example, that Alice and Bob successfully swap BTC for ETH respectively during an Epoch but wish to keep trading during that same Epoch. Alice will have no issue trading her newly acquired ETH during the current Epoch, since there's nothing stopping her from sending trade requests to S, getting matched, and conducting offline trading. The smart contract recognizes that Alice is now the current owner of those funds, so any subsequent user can attempt to make valid claims on those coins within the current cycle. Say Alice trades those coins, or a portion of them, to Charlie. Charlie can turn around and do the same as long as it is still the same Epoch phase. Each continual trade of a coin will increment its nonce, so even if both Alice and Charlie submit claims for that coin in the Claim phase, the contract will only recognize Charlie's as valid. However, for those receiving BTC during the swaps this is not possible as these transactions don't have the same logic that Solidity can support for ETH smart contracts. If Bob wants to keep trading what he received from Alice (assuming it all went through and he has a doubly-signed tx ready for him to claim the 1 BTC after the timelock), he will need to create another timelocked, 3 of 3 multisig between himself, Tesseract, and the other party and engage in the same process as above. Only in this case, he is spending an unconfirmed transaction, since it is prior to the initial trade's timelock and he hasn't broadcasted the tx that sends him the 1 BTC. So the 1 BTC that he deposits to the multisig script doesn't really exist yet. This could lead to an issue if for some reason Bob's redeem transaction isn't accepted by the Bitcoin blockchain as all future swaps of that Bitcoin value will be invalid. Thus, continual swapping poses an issue for Bitcoin in this current protocol.

This service is largely a proof of concept. We chose to work solely with ETH and BTC to test the protocol in an example scenario. However, in reality we would want to expand this to other cryptocurrencies such as Litecoin, Bitcoin Cash, ZCash, and others. Each cryptocurrency will have slightly different challenges for supporting the necessary logic to run the protocol (for instance, implementing multisignatures and timelocks might be different) so we would need to write an API for each additional cryptocurrency. Although we haven't written this API for Bitcoin, we have successfully facilitated the swaps via raw transactions running on a Bitcoin test client. Extending these copy and paste transactions into an automated API is not far-fetched, and is promising for other cryptocurrencies since Bitcoin is relatively rigid and locked-down.

Due to time constraints we were not able to write the API for Bitcoin, which would have allowed us to test the scalability and efficiency of this protocol. Some metrics to look at would be throughput, overall latency, and transaction fees (related to gas consumption) for various trade volumes and frequencies. We also would need to do the same tests with the assumptions of malicious actors. Lastly, the sizes of the Epoch, Claim, and Withdraw phase need to be agreed upon by all cryptocurrency networks. For testing we hardcoded this into our Ethereum smart contract with values 10, 10, and 3 blocks respectively. Changing the Epoch size will allow more trades to occur but increase the amount of work needing to be handled by the smart contract during Claim and Withdraw, which inevitably leads to higher gas requirements. We would therefore want to test how tuning these parameters would affect scalability.

Section 6: Conclusion

In conclusion, we successfully removed Tesseract's reliance on a trusted clock, while preventing eclipse attacks and maintaining the necessary real-time feature. For efficiency, all of our trading process is done off-chain instead of recording each trading transaction on-chain which requires significant verification time. More testing can be done to support this conclusion, however we are confident in its potential. For security, the timelocked multisignature contracts guarantee an honest user never loses their initial deposit even though the server or the other trading party are malicious or unresponsive. In addition, we let users split their traded amount into multiple micropayments. Both parties transfer their token only when they receive the token from the other party. Thus, if one party is malicious or unresponsive (they haven't held up their end of the deal), the honest party will at most lose the value of their most recent microswap which is negligibly small (as determined by the user). Moreover, in order to let users further trade the money received from other users, we use nonces to identify the validity of each coin. It will be incremented whenever this coin's owner changes, so only the latest owner has the right to claim and eventually redeem this coin.

We started this project by reading several papers and analyzing their advantages and disadvantages. Afterwards, we learned how TEEs, timelocked multisignature contracts, and off-chain micropayment channel works and came up with a solution that preserves the security and efficiency of Tesseract while removing the reliance on a trusted clock. During implementation we learned how to build an active server and client in Rust (an SGX enclave supported language) that can send and handle requests, and hash and sign messages. We found that the compiler of

Rust is very strict about the lifetime, scope and mutability of each variable. Though it seems annoying when fixing those errors, it really helps increase the security and robustness of our code. We also learned how to implement multisignature contracts for Ethereum and Bitcoin using Solidity and Bitcoin Script respectively. These maintain most of the logic for the bi-directional channel, verify signatures, and allow deposits and withdraws.

For future work, we wish to combine the features of Obscuro [4] in order to gain anonymity. In addition, we need to make sure our server has enough scalability so that it can support more clients. Lastly, we should implement a convenient User Interface for the client side that abstracts all the responsibility away from the end user and is implemented behind the scenes via APIs for each supported cryptocurrency. These APIs should carry out the bidirectional, timlocked, and multisignature logic specified in our protocol and interact with the corresponding cryptocurrency networks. These changes will allow us to upgrade our system into a real-world application for users in the market to use.

Section 7: References

[1] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, Ari Juels. “Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware”. In *IACR Cryptology ePrint Archive*, 2017.

[2] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson Ari Juels, Andrew Miller, Dawn Song. “Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts”. In *ArXiv*, 2018.

[3] Ethan Heilman, Foteini Baldimtsi, Sharon Goldberg. “Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions”. In *IACR Cryptology ePrint Archive*, 2016.

[4] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, Prateek Saxena. “Obscuro: A Bitcoin Mixer using Trusted Execution Environments”. In *IACR Cryptology ePrint Archive*, 2017.

[5] Federico Tenga. 2018. *Understanding Payment Channels*. [ONLINE] Available at: https://blog.chainside.net/understanding-payment-channels-4ab018be79d4. [Accessed 5 February 2018].

[6] Poon, J., and Dryja, T., 2016. https:// lightning.network/lightning-network-paper.pdf.