ETHERSHIFT
WHITEPAPER

Taylor Hornby & Josh Case
Ethershift is a service that allows users to quickly and easily exchange between different ERC20 tokens at the market rate minus a fee, without having to go through the hassle of signing up at an exchange, or learning how to use a decentralized exchange like IDEX. Decentralized exchanges like IDEX are amazing, however a lot of average crypto currency investors feel like they are too complicated, and therefore a lot of great projects never get the exposure they deserve. Ethershift is making the process of acquiring the best and up & coming tokens easier than ever before.

Ethershift has no accounts, collects 0 customer data, and does not do KYC.

Here’s how Ethershift works:

To exchange tokens, users send value to a special address (which indicates the type of token they would like to receive back) and the exchanged value will be sent back to them at the address they sent the original tokens from. This user experience is implemented by fulfilling the exchange using hot wallets, and then later mirroring the trade (at a hopefully equivalent rate) on an exchange, and periodically re-balancing the exchange/hot wallets. Ethershift takes a cut of profit (half of it in the form of the input token, half of it in the form of the output token) to account for the risk of the exchange rate changing. Third-parties are able to integrate Ethershift into their own systems and receive a 50% cut of the profit via the Ethershift API.

The back-end service is an application written in Node JS, running on Amazon EC2. To access the Ethereum network, a full node is installed on the server, and the back-end service communicates with it through its JSON-RPC API. The service uses an SQL database to keep track of profits.
The service will provide the following API to the front-end website:

1. **ListSupportedERC20Tokens()**: Returns a list of supported ERC20 tokens and “ETH”.

2. **GetNewConversionAddress(token_name, profit_share, profit_address)**: Generates a new Ethereum address for converting value into the token with name `token_name`. A profit_share (value between 0 and 1) fraction of the profits generated by this address will be set aside to be sent to profit_address once the value crosses some threshold.

3. **GetThirdPartyProfitBalance(address)**: Returns the unsent profit-sharing balance earned by the address.

4. **GetHotWalletBalances()**: Returns the current hot wallet balances for each token.

5. **GetExchangeBalances()**: Returns the balances for each token at each exchange.

The service will have the following configuration options:

1. A list of supported ERC20 tokens plus “ETH”.

2. **HardwareWallet**: An address where profits are sent once they surpass a threshold.

3. **ProfitSendThresholds[ERC20 or “ETH”]**: A threshold value for each supported ERC20 token (and “ETH”).

4. **ProfitCut**: A value between 0 and 1 specifying the fraction to be saved as profit.
5. **ThirdPartyProfitCut**: A value between 0 and 1 specifying how much of the overall profit gets sent to third parties (with the remainder going to Ethershift).

6. **ThirdPartyProfitSendThreshold[ERC20 or “ETH”]**: A threshold for each ERC20 token and ETH. Third parties’ profits get sent out once they cross these thresholds.

7. **IdealHotWalletAmount[ERC20 or “ETH”]**: An ideal amount of each token to keep in the hot wallet and exchange wallets. The system will keep the actual amounts between half and 1.5x the ideal value. The system will be seeded with this amount of each token once for the hot wallet, and again for each exchange. So, to support N2 exchanges, the system needs N times the ideal amounts of each token. The ideal amounts is based on the observed trading patterns, so that the exchange/hot wallets don’t get depleted faster than they can be rebalanced.

The database will store (among other things):

1. A list of Ethereum addresses, and for each one:
   a. The name of the ERC20 token (or “ETH”) that any value sent to this address should be converted into.
   b. For each ERC20 token (and “ETH”), the current balance of the third-party’s profit earned by the address.
   c. For each ERC20 token (and “ETH”), the current balance of Ethershift’s profit earned by the address.
   d. The third party’s address that profits will be sent to.
We organize the back-end’s implementation is as follows:

Exchange Objects (1 or more instances)

Exchange objects are abstractions of an ETH/ERC20 token exchange. They control the balance stored on those exchanges and are the path through which the rest of the application makes trades, deposits, and withdrawals. An Exchange object provides the following API.

* GetBalance(token): Returns the balance of the token held at this exchange.

* EstimateRate(tokenA, tokenB, tokenAvalue): Conservatively estimates the amount of tokenB that can be bought with tokenAvalue of tokenA in the near future. Returns an error if there isn’t enough tokenA balance on the exchange to actually carry out the trade. The internal implementation buffers trades before they are made, so it will take those outstanding orders into account. It will also take into account the exchange’s deposit/withdrawal/trading fees and gas prices, so that it represents the rate you would get if you deposited tokenAvalue of tokenA into the exchange, made the trade, and then withdrew it (paying for gas and possible fees to the exchange in each of these steps).

* MakeTrade(tokenA, tokenB, tokenAvalue): Using the balance on the exchange, buys tokenB with tokenA at the best rate currently available, returning the amount of tokenB that was bought. Of course there can be no guarantee, but every effort should be made to ensure the exchange rate is better than what was returned by EstimateRate(), as long as EstimateRate() wasn’t called too far in the past. Despite the effort, the trade might still happen at a worse rate. To be safe we have multiple asynchronous calls happening at once.

* Other functions for re-balancing the wallets.
All of the exchanges’ balances will be seeded with an ` IdealHotWalletAmount` of each token. The wallet on the server will be seeded with the ` IdealHotWalletAmount` amount of each token. A service watches for tokens being sent to the Ethershift addresses (waiting for at least 12 confirmations. For each of these transactions, it executes all of the following actions before processing any more deposits:

1. Identifies the desired output token type, based on the address the tokens were sent to.

2. Keeps a `ProfitCut/2` fraction of the value in the hot wallet, and updates the profit balances in the database (according to `ThirdPartyProfitCut`).

3. Asks all of the exchange objects for their exchange rate estimate between the desired pair of tokens and amount sent in, and picks the best one.

4. Launches an asynchronous `MakeTrade` call on the Exchange’s API.

5. Sends the expected output of the trade back to the user (at the address they sent the original tokens from) from the hot wallet, less a `ProfitCut/2` fraction, and updates the profit balances.

If the call to `MakeTrade` returns failure, then it goes back to step 1. If the call returns failure more than 3 times in a row, it sends an email alert for manual intervention. If the call succeeds and the rate is worse than what was provided to the user, it sends an email alert.

It’s important that steps 1–4 are not run in parallel for different deposits into Ethershift. This is necessary to get the safest exchange rate estimates, because `EstimateRate` may make use of outstanding calls to `MakeTrade`. 
It’s possible for there to not be enough balance in the hot wallet or the exchange wallet for these steps to go through. If that happens, it sends an email alert (it’s an indication that IdealHotWalletAmount is too small for that token), and add the transaction to a queue to be retried later.

The Exchange object’s EstimateRate includes all of the costs of depositing into the exchange, making the trade, and then withdrawing the trade. So, Ethershift will still be profitable even if every trade triggers an individual deposit and withdrawal into the exchange. However, gas can be saved by waiting until any of the wallets fall below their IdealHotWalletAmount/2 or rise above their IdealHotWalletAmount*1.5.

If there is only one exchange being used for any given coin, then (assuming not too much variability of the exchange rate) the balances in the exchange will mirror the balances in the hot wallet: if the hot wallet gives away 1ETH for 5TOKENS, the hot wallet will be sitting at (-1ETH, +5TOKENS), and after the exchange goes through the exchange’s balances will be at (-5TOKENS, +1ETH). With one exchange, the hot wallet’s balances will drift too far away from the ideal amount precisely when the exchange’s balances do, too. And so, with only one exchange, it’s easy to guarantee that no deposit or withdraw to/from the exchange is ever less than IdealHotWalletAmount/2. If this were the case, then the exchange’s EstimateRate could provide a better rate by assuming that deposits and withdrawals are batched into transactions with minimum amount IdealHotWalletAmount/2 (since this would use less gas than making a deposit/withdraw for each trade).

When there is more than one exchange, the trades will be distributed over multiple exchanges. It’s possible for one of the hot wallet’s balances to drift far away from the ideal value while all of the exchanges’ balances are still within an acceptable range. The way to bring the hot wallet’s balance back near to the ideal value is to make deposits and withdrawals that can
be smaller than half of the ideal value. In order to avoid even more value-shuffling between the different exchanges, the hot wallet re-balancing comes from the same exchange that the trade happened on. So, in the worst case, N small trades totaling IdealHotWalletAmount/2 are distributed over all N exchanges. The hot wallet then only needs to be rebalanced, which would require N deposits and withdrawals, one for each trade. This is why, to support multiple exchanges, EstimateRate needs to factor in the full cost of depositing, trading, and withdrawing.

It’s also possible for an exchange’s balance to drift too far while the hot wallet’s balances are still in the acceptable range. This would happen when, for example, ExchangeA provides the best rate when buying TokenB with TokenA and ExchangeB provides the best rate when buying TokenC with TokenB. If the incoming orders for TokenA→TokenB roughly match up with the incoming orders for TokenB→TokenC, then the hot wallet’s TokenB balance will remain in the ideal range, but ExchangeB will develop a deficit of TokenB.

As we progress, we will do a lot of fine-tuning of the IdealHotWalletAmounts and the algorithms for distributing trades and re-balancing the wallets to find the most profitable middle-ground between “we’re forced to use one exchange’s rate but deposits and withdrawals will always be amortized” and “we can pick the best exchange rate from any of N exchanges, but we’re reducing the amount of deposit/withdrawal amortization.” The Ethereum wallet is seeded with some ETH to pay the gas for the transactions. The back-end service sends notifications to an email address when the balance is running low and needs to be replenished.
A separate EC2 instance will host Ethershift’s user-facing website. This website will show users the addresses (with QR codes) they should send tokens to in order to make exchanges. It will also have an affiliate account login system where users can log in and generate special addresses that will send them a cut of the profit anytime someone uses it. It will also allow access to the Ethershift API.

The websites has an administration panel allowing some visibility into what’s going on in the Ethershift backend server. For example, it can use GetHotWalletBalances() to show the current hot wallet balances, and other APIs will be added to view the transaction history, etc. The administration panel is a read-only view of what’s happening on the back-end, and will not be able to manipulate the back-end.

Since Ethershift fulfills its users’ trades before it can be certain of the exchange it will actually get, there is a significant risk that the exchange rate will worsen between the time Ethershift fulfills the user’s trade and when the Ethershift back-end makes the corresponding trade on an exchange. It’s also possible for the exchange rate to improve and it might even out in the long run, but nevertheless Ethershift should assume some of its cut of the profit will be going towards paying for these unprofitable exchanges. The back-end detects whenever it happens and make metrics available so that the exchange rate estimation algorithms can be improved.
The Ethershift servers and accounts (e.g. domain registrar, AWS, etc.) are set up and administered from a new laptop running Linux that is used exclusively for this purpose. The passwords to the accounts are randomly generated and stored on this machine. This helps prevent malware and criminals from gaining access to the servers. No software development, web browsing, or email viewing happens on this machine. The SSH private keys for accessing the servers, and the Tarsnap backup secrets, are generated on this machine and never leave it except in the backups. At least two encrypted backups of this laptop should be kept somewhere safe, with one of them off-site. We periodically check (using the administration laptop itself) that the backups are working properly and contain all of the important information. To reduce the threat of physical theft, we use full-disk encryption with a memorized password on the laptop, and fully power it off whenever it is not being used.
To reduce the risk of monetary loss, it is crucial for Ethershift’s design and implementation to undergo third-party security audits. Subtle bugs in the back-end service could result in downtime or token theft. Extra care must be taken to harden servers against attack. Full details on how we plan to harder our server are outside the scope of this document, but some general points can be made:

* Two-factor authentication on all accounts (AWS, the domain registrar, etc.).

* Each system is running on its own dedicated hardware, to avoid possible theft from cross-VM side-channel attacks.

* All software on the servers, including the Ethereum node, must be updated regularly.

* Disabling of all unnecessary services on the servers.

* Don’t run anything as root unless it’s absolutely necessary.

* Administer the servers with SSH, and turn password authentication off so that the only way to get in is to have the SSH private key. Only administer the servers from the administration laptop.
**Ethershift Token**

The Ethershift token is an ERC20 token that represents 50% of the gross fees collected by Ethershift. Each transaction that is made on any of the Ethershift products yields a 5% fee. The system automatically calculates the gross fee Ethershift collected after gas, and splits it into 2 wallets. One wallet is used for operating costs, salaries, legal, etc, and the other wallet is used to airdrop to ESH holders.

Starting in 2019, each quarter 50% of all gross fees collected by Ethershift will be airdropped to ESH holders.