CHARON YELLOW PAPER

INTRODUCTION

Charon is a privacy system that utilizes shield transactions through a cross-chain asset, the charon dollar (chd), created through a privacy enabled, cross-chain automated market maker (PECCAMM). It works by having AMMs on multiple chains, with one of the assets on each AMM being the charon dollar, a free-floating token created by deposits on an alternate chain and then a withdrawal via a zero-knowledge proof[i][1]. To achieve this cross-chain functionality, charon utilizes an oracle to pass commitments (proof of deposits) between chains. It achieves privacy by breaking the link between deposits on one chain and withdrawals (on the same or another).

For a simple introduction to charon and the base mechanisms, the whitepaper and tokenomics paper can be found here: https://github.com/charonAMM

CHARON FUNCTIONS AND DETAILED DESCRIPTIONS

Definitions:

AMM – automated market maker[2]. An exchange structure where parties provide passive liquidity to two (or more) tokens that can be traded at the price derived from the constant calculated when balancing of the two (or more) sides of liquidity.

baseToken – the token used in a given chains system. For each chain (e.g. Ethereum, polygon, gnosis), the AMM has two tokens, a base token and chd. The baseToken for each chain is the wrapped native asset. Note that if deploying a new charon system, the rebate function relies on the pricing of the gas token via the price derived from the AMM (baseToken x chd).

LP – (Liquidity Provider) – someone who stakes CHD and the baseToken or just chd in an AMM for a pool token and a share of the fees.[3]

[1] zero-knowledge proof - https://en.wikipedia.org/wiki/Zero-knowledge_proof
[2] https://www.coindesk.com/learn/what-is-an-automated-market-maker/
[3] https://www.gemini.com/cryptopedia/liquidity-provider-amm-tokens

Functions:

Not including getters, internal, and initializing functions, there are only 8 functions in the charon system.

- addRewards
- depositToOtherChain
- lpDeposit
- lpSingleCHD
- lpWithdraw
- oracleDeposit
- swap
- transact

Each function contributes to the overall system and is used for either becoming an LP, trading, or transacting privately.

addRewards – this function is primarily called by the charon fee contract. The purpose of the function is to add rewards in the system. It takes 4 inputs (_toUser, _toLPs, _toOracle, and _isCHD). The first three are the amount of a given token (baseToken or CHD depending on final variable) to go to each type of incentive in the Charon system. The user rewards are paid out when a user calls depositToOtherChain, the LP rewards are simply added to the recordBalance and recordBalanceSynth (sides of AMM pool, the former being the base token, the latter being CHD), and oracle rewards are paid to the parties involved in calling oracleDeposit (e.g. caller of that function and tellor reporter).

depositToOtherChain – this is one of the core functions of the charon system. It takes in a baseToken or CHD, locks it with a proof, and then sends the proof to the other chains. In addition, user rewards are paid out during this function call.

The function takes 4 arguments: _proofArgs, _extData _isCHD,_maxOut

The proof arguments are defined as the proof struct:

```
Proof {
        bytes proof;
        bytes32 root;
        uint256 extDataHash;
        uint256 publicAmount;
        bytes32[] inputNullifiers;
        bytes32[2]
}
```

proof - is the result of the groth16.fullProve() function from the snarkjs library[4] using the corresponding wasm and zkey.

extDataHash - is the keccak256 hash of the second input to the function extData. This hash prevents relayers from manipulating the underlying extData.

publicAmount - is the amount of tokens being transacted publicly (minting or burning private txn value) minus the fee. So a deposit would have a positive amount, but if you are just transacting in the shieled state, this would be zero.

input nullifiers - are a list of the nullifiers (hash of index, signature, chaindID) of the inputs.

output commitments - are a list of the commitments in the outputs (hash of amount, chainId, public keypair, blindings)

The extData arguments are defined as the extData struct:

```
struct ExtData {
        address recipient;
        int256 extAmount;
        uint256 fee;
        uint256 rebate;
        bytes encryptedOutput1;
        bytes encryptedOutput2;

}
```

recipient – is the address of the party receiving the minted chd. Only non-zero if transacting to remove from shielded state

extAmount – this is the same as the public amount, but without consideration for the fee

fee – the fee given to the relayer (in chd)

rebate – amount of base token (gas, priced in chd) taken from the relayer and given to the recipient (e.g. places gas in the wallet so you don't need to fund a new wallet)

encryptedOutput1 / encryptedOutput2 – using the public encryption key of the recipient, the amount of the transaction is packed with a random number and

[4] https://github.com/iden3/snarkjs

encrypted. Since it is a utxo structure, all inputs must be spent, so the output is usually what you send to one person and then what you return to yourself of the inputs.

For the final two variable, isCHD refers to whether the token being deposited is chd or not and then the final maxOut is the limit to the amount of base tokens to be used in case isCHD is false. For maxOut, there were front running concerns where a depositing party could get less chd on the other chain than expected, so this variable allows them to specify the worst rate they're willing to get.

When deposited, the chd or base token is transferred to the contract. The amount of chd to be minted on the other side is extAmount, and the aount of base token necessary for this input is calculated as if you were trading on the AMM (calcInGivenOut). If base token is put into the protocol, the amount in is added to the recordBalance (base token side of the AMM). If CHD is input, it is burnt (sent to a privacy state on another chain). User rewards are then calculated as the userRewards/1000 in the system. The deposit details are then sent to all other chains in the system and specified in the oracles list of the given chain's charon contract. As a final step, the details of the deposit / proof are placed in the merkleTree of that contract to even further increase the anonymity set.

lpDeposit – This function takes three arguments (_poolAmountOut, _maxCHDIn, and _maxBaseAssetIn). Similar to other AMM's this function allows to be LP's to deposit chd and the base token in exchange for pool tokens (shares of pool assets). The _poolAmountOut variable is how many pool tokens are desired for a given amount of chd (capped by _maxCHDIn) and base tokens (capped by _maxBaseAssetIn).

--Single token (chd) input/output price dynamics –

Becoming an LP with only one asset (chd) is an important piece to the Charon system. For pricing charon, the base (or lowest) value is with every chd locked in the AMM's. When new baseTokens are deposited to mint CHD, the CHD could be instantly placed on the other side of the AMM, thus ensuring the price stays equal and CHD acts mechanically identical to a multi-collateral AMM pool token. In reality, not all CHD will be locked in the AMM. any demand for CHD outside of solely LP'ng it, results in CHD going up in value (it's priced in terms of the base asset). This means that the more users who demand CHD in it's shielded state, the higher the price of CHD.

In order to bring the price down, normally you would just swap chd for the base token. This however is different in the charon system. Any time the baseToken is withdrawn, chd is burned (ensuring that minimum value). So a swap of CHD for the base token does not lower the price as much as on a normal AMM. To allow for parties to lower the price of chd(or in future versions of Charon, incentivize the LP'ng of the single token), they can simply deposit CHD on one side of the AMM and receive a corresponding amount of pool tokens. Note that since swapping base tokens for chd does work like a traditional AMM, single sided deposits with the base token are not permitted and single-sided withdrawals of any kind are also not permitted.

The other notable difference to single-chd sided LP deposits vs a normal LPDeposit is that the deposit is amortized over 1000 system events. In specific terms, say you have a deposit of 1000 chd, the 2000 chd is added to the pool in an increment of 2 chd (amountIn/1000) over the next 1000 events (anytime the functions addRewards(), lpDeposit(), lpWithdraw(), _swap(), transact(), or checkDrip() are called). This is to ensure that an attacker can not manipulate the price of chd downward, mint themselves chd with a smaller deposit, and then withdraw the chd (the withdraw is also locked for a day). The slow drip does not prevent this kind of attack, but simply democratizes the price move, allowing MEV/arbitrage to drive the profit towards zero. There is an added benefit here that the scheduled price move will incentivize trades, thus increasing LP rewards.

lpSingleCHD- this function takes two arguments, _tokenAmountIn and _minPoolAmountOut. The two variables are rather selfexplanatory, with the first being the amount of CHD to be deposited and the latter being the minimum pool tokens you need out (a variable to prevent front running attacks). When the chd is deposited, the poolAmount is calculated based upon the pool and chd supplies in the AMM, the chd is added to the singleCHDLPToDrip (a variable that has the total amount of chd to drip over the next 1000 blocks) and the dripRate is calculated (singleCHDLPToDrip/1000). Pool tokens are then minted and the depositor is locked from withdrawing their pool tokens for the next 24 hours.

lpWithdraw – this is the standard withdraw function for LP's in that it returns equal amounts of both the base token and chd. It takes three arguments (_poolAmountIn, _minCHDOut, and _minBaseAssetOut), with the variables being the amount pool token sent in, the minimum amount of chd you need out and the minimum amount of the base token you need out, with the latter two solely as price checks to prevent front running. In addition to sending out the tokens, the pool tokens are burned and the standard fee of 60 bp is charged on withdrawal.

oracleDeposit- an oracle deposit is the process of grabbing a proof, generated from a depositToOtherChain function call, from a connected charon contract on another chain. It takes two arguments, oracleIndex and inputData. The first corresponds to which chain you are pulling from, if your system is connected to multiple. The second is the specific inputData for this call, which ultimately depends on the chains connecting. For example, for the Ethereum-gnosis[5], polygon connections, Charon uses Tellor[6]. The inputData in this case is the depositId from the connected chain, and the oracle contract does the rest. Note that for tellor usage in chains, there is a 12 hour wait between when the tellor report is submitted (right after the depositToOtherChain call) and when it can be placed on the other chain.

Upon pulling the proof from the oracle, the proof is placed into the merkle tree and chd is now in a shielded, private state, ready for parties to call transact() in order to send chd anonymously. After being placed in the merkleTree, oracle rewards are paid to the caller of the oracleDeposit function and to the reporter (in the case of tellor).

Swap – the swap function is the basic AMM transaction where you input tokens from one side of the pool and exit with the other. The only caveat from normal AMM's is that the math and function is adjusted so that chd is burned when input instead of simply being placed on one side of the AMM.

The function takes four arguments (_inIsCHD, _tokenAmountIn, _minAmountIn, and _maxPrice). The self-explanatory inputs tell whether the input is chd (baseToken if false), the amount of that token being sent in (be sure to approve the function before trading), the minimum amount of the other token you need out and the maxPrice after the transaction.

A fee is also taken and sent to the CFC during the swap.

Transact – a transact is when you send chd (or withdraw it from) a shielded state. If withdrawing from the private state, the proof would specify zero outputs and the output would instead be a minting of CHD tokens (conversely inputs have no input to the UTXO but only an output with a deposit). Rebates and relayer fees are also paid on submitting the transact function. Rebates are payments of gas to a wallet. You add it in the extData and the fee (payment to relayer) must be larger than the rebate. The actual fee to the relayer is the fee – rebate (amount they pay in gas, calculated based on market (chd to baseToken AMM swap) price).

In order for parties to get the input to the proof, they can scan a given chain for NewCommitment events (created when transact or oracleDeposit happen). The encryptedOutput's described in the extData are emitted and can be decrypted by the receiving party alone. If your key can decrypt the output, you can derive the necessary information to use the output as an input in your transact.

[5] https://www.gnosis.io/
[6] www.tellor.io

MATH BEHIND CHARON FUNCTIONS

Constants:

$$B = 10^{18}$$

$$Max_{in} = \frac{B}{2}$$

$$Max_{out} = \frac{B}{3} + B$$

Let:

$$t_{bo} = token\ balance\ out$$

$$t_{ao} = token\ amount\ out$$

$$t_{ai} = token\ amount\ in$$

$$t_{bi} = token\ balance\ in$$

$$p_s = pool\ supply$$

$$p_{to} = pool\ tokens\ out$$

$$f = swapfee$$

$$sp = spot\ price$$

calcInGivenOut:

Given the pool’s balance of the token coming in, the pool’s balance of the token coming out, and how much of a token is expected to come out of the pool, this calculates how much of the other token must enter the pool. Finds the ratio of the pool’s balance of the token coming out to the difference in that balance and the amount of token coming into the pool, multiplies that by the pool’s balance of the token coming in, and adjusts for fees.

$$t_{ai} = \frac{t_{bi} * (\frac{t_{bo}}{t_{bo} - t_{ai}} - B)}{B - f}$$

calcOutGivenIn:

Given the pool's balance of the token coming in, the pool's balance of the token coming out, and how much of a token is coming in, this calculates how much of the other token is expected to come out of the pool. Finds the ratio of the token balance coming into the pool to the token amount already in the pool adjusted for fees and multiplies that by the amount of tokens coming out.

$$t_{ao} = t_{bo} * (B - \frac{t_{bi}}{t_{ai} + \left(t_{ai} * (B - f)\right)})$$

<u>calcPoolOutGivenSingleIn:</u>

Given a certain token's balance in the pool, how much a user is sending into the pool, and the total supply of pool tokens, this calculates the amount of pool tokens that will be distributed for the deposit. Finds the normalized ratio of the new token balance to the token balance before depositing and multiplies that by the amount of pool tokens in supply to calculate the updated supply of pool tokens, giving the number of pool tokens to be distributed.

$$p_{to} = \left(\frac{\left(t_{bi} + (t_{ai} * B)\right)^{.5B}}{t_{bi}} * \ p_s\right) - p_s$$

<u>calcSingleOutGivenIn:</u>

Given the pool balance of a token to be withdrawn, the total supply of a different token to be deposited, and the amount of that token getting deposited, this calculates how much of the token will be withdrawn. Finds the normalized ratio of the token supply after the swap to the token supply before the swap and adjusts for fees.

Where token in = pool token:

$$t_{ao} = (t_{bo} - \left(\frac{t_{bi} - (t_{ai} * B)^{2B}}{t_{bi}} * \ t_{bo}\right) * .5B * f)$$

Where token in = base token/chd (used for swaps):

$$t_{ao} = t_{bo} - (t_{bo} * (B - \frac{B}{t_{bi}})^{t_{ai}})$$

calcSpotPrice:

Given a pool's balance of two tokens, this calculates the spot price. Finds the ratio of the balance of the token coming to the balance of the token coming out, adjusting for fees.

$$sp = (\frac{t_{bi}}{t_{bo}})(\frac{B}{B-f})$$

ZERO-KNOWLEDGE CIRCUITS

The cryptographic functions for off-chain use are implemented in the circomlib library. Much of the circom and solidity implementations of the Merkle Trees are taken from tornado cash nova[7] with modifications to fit the charon system, namely the additional checks relating to privately verifying that the proof corresponds to the correct chain. The Solidity implementation of the Poseidon[8] hasher is created by Iden3[9] and the full script to make the contract is in the charon repo, important to note since the contract cannot be verified by traditional block explorers. The SNARK keypair and the Solidity verifier code are generated by the authors using SnarkJS, a tool also made by Iden3.

The two main circom files of note are transaction2 and transaction16 which when built create the corresponding Verifier2 and Verifier16 contracts which are called by the main charon contract. The only difference in the two files is the number of inputs. Note that all transact functions (and ergo proofs) must fit to either have two or sixteen inputs. Inputs of zero should be placed in circumstances where there is not that exact number.

The reasoning behind this is that the underlying circom files create a UTXO structure. A utxo structure takes inputs and maps them to outputs. So if A wants to pay B one dollar, he takes an input that has at least that amount (e.g. C gives A two dollars) and then creates outputs (A gives B one dollar AND A gives A one dollar). Note that there are always only two outputs and the full amount of the inputs must be spent (like all UTXO models). The reason that 2 and 16 are chose is simply following the tornado cash nova standard set. Eventually the system should be rewritten to allow a variable number of inputs, but the 2 and 16 easily cover most uses.

CHARON FEE CONTRACT

[7] https://nova.tornadocash.eth.link/
[8] https://www.poseidon-hash.info/
[9] https://iden3.io/

The charon fee contract (CFC) is a distributor contract on each chain. Revenue from various tokens come into the contract and are distributed as incentives to the charon system. The system will be initialized with a .6% fee on all trades, a number in range with other amm systems. 50% of this fee be given directly to LP's and the other 50% will go to the CFC. This will be the only revenue on most chains, however on the CIT's chain, 100% of the auction proceeds will also be distributed to the CFC.

Initial distribution (can be changed in constructor)
- 10% oracle incentives
- 20% yield farming rewards
- 20% usage farming rewards (to bolster larger anonymity sets)
- 50% staking rewards given to CIT holders

Staking rewards are given to holders on each chain based upon their CIT balance at the start of each month. Since CIT is only on one chain, a snapshot of balances is taken on the first of each month and then passed over to alternate chain CFCs for distribution .

Token Snapshot Methodology: The charon fee contract repository (https://github.com/charonAMM/feeContract) contains scripts to calculate the merkle root of the balances to be used as a snapshot of all CIT balances[10] at a given block (specificied when a fee round is ended (monthly)). To build a merkleTree, all accounts and balances are calculated using in the system by scraping all transfer events for the token (for relevant accounts) and then checking their balances. Accounts are then sorted and then the keccak256 hash of the abi encoded address and balance are inserted into a merkleTree. The root of this merkleTree is then submitted to each chain in the system using the tellor oracle system. Users also need to run the script to generate a location of their balance in the merkleTree and submit the location along with their balance and address to the contract to claim.

Expiration of Rewards: Rewards are rolled back into the contract after 4 periods (months). All unclaimed rewards are added as fees back to the network and distributed proportionally.

CHARON INCENTIVE TOKEN

The charon incentivize token (CIT) is a token associated with the charon system. Its purpose is to bootstrap usage of the system and promote activity such as oracles, keepers, liquidity providers, and usage subsidies to foster a larger anonymity set. Starting at zero tokens, a set

[10] https://ethresear.ch/t/erc20-snapshot-using-merkle-proofs/4423

number of tokens are minted each week for perpetuity (e.g. 2,000). The distribution of these tokens is determined by a public auction that ends each week. The proceeds from this auction will be sent to the charon fee contract on the CIT's chain.

The auction: Each auction is as simple of an auction on Ethereum as could be built. There is an end date, a top bidder, and a bid amount. New bids are accepted until the end date and if the bid amount is greater than the previous top bid, there is a new top bidder and the bid is stored in the contract. At the end of the auction, a function (startNewAuction()) can be run which mints new tokens to the top bidder, sends the top bid to the fee contract, and starts a new auction by adding 7 days to the time when this function is run.

Purpose of distribution mechanism: CIT distribution is modeled after the "fair" pow distribution, namely that there is a recurring auction for new tokens, available to all. Rather than auction new tokens using electricity via hashpower however, the CIT is auctioned via a traditional auction on Ethereum. The benefit of this method is that we get similar distributional effects to pow without the waste as resources (the bid in tokens) are cycled back into the system in the form of fees to the charon fee contract.

CONCLUSION

For more information on Charon: www.github.com/chaonAMM