zk-DEX: Private and Decentralized Exchange Protocol

Onther Inc.

Nov, 2019

Abstract

zk-DEX is a privacy-focused decentralized exchange protocol based on zk-Dai[2]. The protocol does not reveal any critical information about the traders by verifying execution of two conditional transfers through zero-knowledge proof[20]. On zk-DEX, users can trade only submitting proof that they have performed the designated process correctly, instead of sending transactions to a decentralized exchange[5] running on the blockchain, which can expose their transaction history. This allows both traders to complete the transaction without exposing any information to a third party.

1 Introduction

Since the birth of Bitcoin, many cryptocurrencies have been created, disappeared, and at the same time, traded. Currently, the total market capitalization of cryptocurrencies is about $200 billion[4], and most cryptocurrencies are traded on centralized crypto exchanges(CCE) operating outside the blockchain[6]. CCE provides users with the convenience of not having to deal with creating transactions on the blockchain while trading assets, and serves as a bridge between fiat money and cryptocurrency. However, since the CCE holds custody of user's cryptocurrency to provide off-chain exchange service, there are constant concerns regarding users not being able to ensure the safety of their assets if there are any issues with the exchange.[7].

For example, in the case of Mt. Gox, about 850,000 bitcoins were stolen when the exchange was hacked. Also, in QuadrigaCX, which was the largest bitcoin exchange in Canada, $1.5 billion worth of customer assets were frozen when the unexpected death of the CEO, who was the only person with access to the wallet, happened.

To solve these issues on a fundamental level, a decentralized exchange(DEX) model[5] was proposed where the users exchange cryptocurrencies peer-to-peer on the blockchain instead of having a third-party settling orders off-chain. This allowed DEX to be free from the issue of custody, which was a fundamental problem in the CCE model. DEX models on the blockchain also improved censorship resistance because a random miner or validator nodes handle transactions[7], while on CCEs the identities of traders are registered and can be censored in advance or after trading occurs.

However, DEX models have two major problems. The first is the scalability problem. Since DEXs operate on top of blockchains, it cannot be free from blockchain scalability issues. Second is the

privacy problem[15], which also results from the fundamental design of blockchain. If the blockchain does not support privacy at the protocol level, all transactions, states, or balances will be revealed to the public. This means all trading history of users in the DEX are revealed, which causes privacy issues.

1.1 This work

This work proposes a new decentralized exchange model, zk-DEX, to solve privacy issues, which is one of the major problems of DEX. In zk-DEX, users convert their tokens to an encrypted commitment called note like in zk-Dai instead of using the token directly. Notes contain information about ownership and value of tokens like the unspent transaction output(UTXO) in Bitcoin[13]. Third-parties cannot figure out such information because notes are only stored as hashes on the blockchain.

When using notes, users first prove their ownership of notes through zk-SNARKs. Trade occurs only with these notes, and the process consists of 3 steps, which are MakeOrder, TakeOrder, and SettleOrder. In each step, they have to verify through zk-SNARKs that they have used and created a valid note under appropriate conditions.

In each step, the users generate zero knowledge proof off-chain and submit to the blockchain. The smart contracts on the blockchain validate the submitted proof. When the proof is successfully verified, a set of notes is sent to the users involved in the transaction. This ensures that no third-party has information about the identity of the trader(e.g. address) and the volume of the transaction processed.

In zk-DEX, transactions go through an order book, a list of orders registered in a decentralized exchange smart contract. On the order book, only the price of orders will be available without the amount. This ensures privacy, while at the same time providing visibility of the price of all orders in the market.

Also, all zk-DEX transactions do not require separate encrypted communication channels. Using a separate communication channel will ensure the same amount of privacy as zk-DEX in the transaction process, but this will burden users with an additional administrative obligation. Instead of adding a separate communication channel, zk-DEX chose to add an extra step on the transaction process.

2 Related Work

As at the time of writing, there are a variety of studies on simple payment protocols to ensure privacy on blockchains which do not support privacy features at the protocol level, such as Ethereum[1]. However, there were not many studies regarding protocols that focus on privacy in exchanging assets. So let's first take a look at previous studies of simple payment protocol, and then at private DEX, which guarantees privacy, presented in ZEXE[**zexe**].

2.1 Private Payment

2.1.1 Hopper

Hopper[3] is a very simple mixer based on Ethereum. It is a simple payment protocol that allows the link between the sender and recipient to be untraceable. To transfer tokens through Hopper, you must first perform the Deposit process. When depositing, users register a specific hash value and deposit as many tokens as they want to transfer to that hash value. The $hash(secret, address)$ consists of the address of the recipient and a secret value that only the sender knows. If the token is deposited correctly, the hash is added to the merkle tree in the Hopper contract. Then, when a user proves, through zero-knowledge proof, what the values contained in the hash are, the token will be sent to the address recorded in the hash value. The important thing is that we cannot specify which hash in the merkle tree were used in this process. To prevent double-spending, the prover must register $Nulifier = hash(secret)$ of the hash in the contract. In the Hopper protocol, the level of privacy depends on the number of users, and transactions can only be made in fixed amounts.

2.1.2 Heiswap

Heiswap[17] is a simple form of a mixer, similar to Hopper. Heiswap also makes the transaction link untraceable using the linkable ring signature[18] and stealth address[16]. If you want to transfer ether through Heiswap, you must deposit first. After depositing ether, the recipient address is registered to ring pool, and hei-tokens are generated. The recipient will be able to withdraw ethers proving that the user is the owner of hei-token through the ring signature[14]. However, Heiswap also cannot guarantee high levels of privacy when only few users are participating in the pool, and users can only send a fixed amount of tokens.

2.1.3 zk-Dai

zk-Dai[2] is another simple private payment protocol based on Ethereum that conceals the identifiers, link, and the amount of token of the sender and recipient. The difference of zk-Dai is that the tokens are converted into a form of notes. The note consists of $[pubKey, value]$, and only the hash value and the encrypted note is stored on the blockchain. To use a note, you must prove through zero-knowledge proof that the hash of the note can be correctly configured, that you own the secret key of the public key, and that you have recorded the correct amount of token. The structure of zk-Dai's notes is very similar to that of Bitcoin's UTXO, and is almost identical to the approach of zcash[**zerocash**] in that it does not expose specific information of the note by using zero-knowledge proof. However, because anyone can identify which note hash was used for the transaction, anyone can track a specific history of each note.

2.1.4 Ethereum 9¾

Ethereum 9¾[10] is a protocol that allows ether and tokens to be sent anonymously on Ethereum, using mimblewimble[8], nullifier, merkle mountain range[12]. Ethereum 9¾ does not reveal identifiers

and links of sender and recipient, transfer amount, as well as the encrypted value used in the transaction (e.g., zk-Dai's note). In Ethereum 9¾, mimblewimble transactions are stored in the merkle mountain range, and to use them, you must prove that you used correct transaction input. In order to prevent double-spending, the unlinkable tag used history is recorded, which replaces an input without exposing information, rather than recording that a particular transaction input was used.

2.1.5 Zether

Zether[**zether**] is a protocol that enables anonymous transfer of assets on blockchain such as Ethereum. It ensures privacy through bulletproofs-based range proof and ElGamal encryption, which is also a type of zero-knowledge proof. Zether is different from others in that it is based on the account model. In most protocols, tokens must be converted into encrypted form(commitment) and are verified through zero-knowledge proof, but it involves burdens of managing encrypted tokens separately. To improve this inefficiency, Zether uses an account-based model.

2.1.6 AZTEC protocol

AZTEC protocol[19] is a private payment protocol on blockchain. Specifically, it does not reveal information about the identities of senders and recipients, and the amount. Users will use the tokens by converting them into an encrypted form, aztec note. Similar to zk-Dai, it verifies that the value of input and output notes are equal through zero-knowledge proof. The important point is that instead of using existing methods such as zk-SNARKs, they self-configured zero-knowledge proof through the combination of homomorphic arithmetic and range proofs. Trusted setup also enables efficient proving and verification.

2.2 Private DEX

2.2.1 ZEXE

ZEXE[**zexe**] is a protocol that can verify the validity of transactions through zero-knowledge proof, thereby hiding all information. Unlike the protocols discussed above, ZEXE differs in that it is a project on ensuring validity and privacy for arbitrary operations, not only for simple payments.

In ZEXE, the state and its state transition functions are expressed in the form of records nano-kernel. Records nano-kernel consists of record, and two randomly defined functions, the birth/death predicate. If a new record is generated using an existing record, it will first verify that the death predicate of the existing record and the birth predicate of record has been executed correctly.

An example presented in ZEXE is a private DEX. The intent-based exchange protocol allows users to trade by exchanging necessary information about orders in advance through a separate communication channel. Therefore, it is difficult to utilize the protocol on the existing orderbook-based DEX.

2.2.2 Voxelot zk-dex

Voxelot's zk-dex[9] is a DEX protocol that leverages zero-knowledge proof to ensure privacy. You can trade with notes that are created in a very similar way to zk-Dai. However, it is required to have a separate communication channel to share information and agree on the conditions of the order. Therefore, this is also difficult to apply to orderbook-based DEX.

2.3 Comparison

Privacy Support Coverage and Efficiency							
Index	Hopper	Heiswap	zk-Dai	Eth 9¾	Zether	Aztec	zk-DEX
Link	✓	✓	✓	✓	✓	✓	✓
Amount	-	-	✓	✓	✓	✓	✓
Identity	-	-	✓	✓	✓	✓	✓
Input	-	-	-	✓	✓	-	-
Proving Cost	Medium	Low	Medium	High	High	Low	Medium

Table 1: Private Payment

Privacy Support Coverage and Efficiency			
Index	ZEXE	Voxelot	zk-DEX
Link	✓	✓	✓
Amount	✓	✓	✓
Identity	✓	✓	✓
Input	-	-	-
Order-book support	-	-	✓
Proving Cost	Medium	Medium	High

Table 2: Private DEX

Remark. *Link, amount, identity, and input in table1 and 2 indicate to what extent the protocol can ensure privacy. Link is the link between sender and recipient, amount is the amount transferred, identity refers to the information of sender and recipient, and input is the encrypted value used for transfer. For Zether, there is no separate input-output structure, so it is considered to ensure privacy about input information.*

Proving cost means the cost of generating proof. This comparison requires the performance to be measured in the same environment, but due to structural limitations, the values were calculated by taking into account the data presented by each protocol and logical complexity. There are three classes: low, medium, and high, and there can be performance variance within the same class. The important thing is that the stronger the privacy, the higher the cost. Given this, Aztec Protocol has shown that the cost of computation is also low, while ensuring an appropriate level of privacy.

Usually, verification cost is considered important as well as proving cost. However, the reason for not being included in table1 and 2 is that the majority of protocols use zk-SNARKs, in which case the

cost of verification is constant and therefore no comparison is meaningful. Also, verification cost is always lower than proving cost in succinct zero-knowledge proof, so bottlenecks arise in proving rather than verification. Therefore, dealing with only proving cost was sufficient to compare each protocol.

3 zk-DEX protocol

3.1 Problem Statement

Let's assume that two traders, Alice and Bob, are trading on a DEX. Alice wants to trade her ether with Dai. Let's say that the price of ether is 200 Dai / 1 ether. Alice wants to trade 10 ether, and Bob wants to trade with 1000 Dai.

The typical trading process for the DEX proceeds as follows: First, Alice will make an order to trade ether with Dai in a smart contract and will set the price at 200 Dai. Simultaneously, Alice will transfer 10 ether to the contact. After checking the order, Bob sends 1000 Dai to the contract to take the order. Immediately, the DEX contract will execute a settlement process of exchanging 10 ether and 1000 Dai for 200 Dai / 1 ether price, and sends 1000 Dai to Alice, 5 ether to Bob and 5 ether to Alice as change.

The problem is that all actions, such as making and taking orders are broadcasted as a transaction in blockchain. These transactions are public to all participants in the blockchain for verification with re-execution, so information about the order will also be revealed. Specifically, those information include 1) identities of traders, 2) trading amount, i.e. the trading volume, and 3) type of token traded are exposed.

3.1.1 Objectives

Here we want to design a protocol for Alice and Bob to trade without exposing their identities or trading volume on the DEX. However, this protocol does not ensure privacy for token types.

3.2 Notes

In zk-DEX, all assets are expressed in the form of notes, and only encrypted notes are stored in the blockchain.

Definition 1. *Note Notes are securities representing the ownership, face value, token type, and etc. Notes consist of five data fields: pk, v, v, vk, salt.*

$$note = [pk, v, type, vk, salt]$$

- pk is the public key of the note owner, used to prove ownership. Denoted as $note.pk$.
- v is the face value of the note. Denoted as $note.v$.
- $tokenType$ is the token type of the note. Denoted as $note.type$.

- vk is the viewing key of the note, which is used to encrypt the note so that only the Taker can see it during the trading process. Denoted as $note.vk$.

- $salt$ is a random value, used to ensure that each note has a unique value. Denoted as $note.salt$.

Definition 2. *Note hash Note hash is a hash of each field value of a note, used to privately store the note on the blockchain.*

$$noteHash = hash(note.pk, note.v, note.type, note.vk, note.salt)$$

Encrypted notes are represented as follows.

$$encNote = encrypt(pk, note)$$

$encNote$ is the value of each data field in the note encrypted by pk. It allows new owners to have exclusive access to note data after trading. $encNote$ is recorded in smart contacts with $noteHash$.

3.2.1 State of Note

The note has four states: $INVALID$, $VALID$, $SPENT$ and $TRADING$. The meaning of each state is as follows.

- $INVALID$: This means that the note has not been created. It is not created and cannot be used.

- $VALID$: Indicates that the note has been created correctly and is available for use.

- $SPENT$: This means that the note has already been used. It can no longer be used.

- $TRADING$: This means that notes are currently being used for trading. It cannot be used until the trading is completed.

3.2.2 Create

To create a note, you must perform the $Create$ process. $Create$ consists of a $Create.prove$ and a $Create.verify$, the former is the process of generating zero-knowledge proof for a note on off-chain, the latter is the process of verifying the proof on-chain. Not only $Create$, but all the processes regarding notes to be described in the future consist of $prove$ and $verify$. $Create$ is configured with Algorithm 1.

$Create$ will verify whether the amount of token deposited at the time of note creation matches the face value of the note, and also whether the type of token deposited matches the token type of the note. If verified successfully, nh and en are recorded and notes are available for use. In addition, the state of the note changes from $INVALID$ to $VALID$. While creating a note through $Create.verify$, information about the owner of the note is not revealed, but during the verification process, the face value and type of that note are revealed.

Algorithm 1: Create

Input:
- private
  n: note to be created
- public
  nh: hash of n
  v: value of n
  t: toktype of n

Output:
p: zero-knowledge proof

Function `CreateProve`**:**
    $nh = hash(n)$
    `// check note hash with note data`
    $n_v = v$
    $n_{type} = t$
    `// check value and type`
    return p

Input:
nh: hash of n
en: encrypted n
v: value of n
t: type of n
p: zero-knowledge proof

Output:
s: bool type value indicating result of verification

Function `CreateVerify`**:**
    $s = verifyProof(nh, en, v, t, p)$
    return s

3.2.3 Spend

Spend is the process of transferring generated notes. *Spend* uses its own $oldNote$ to create two new notes, $newNote1$, $newNote2$. $newNote1$ is used to transfer notes, and $newNote2$ can be used primarily to return the remaining balance after transfer. Of course, it is also possible to send the note to two users using all of the note's face value. *Spend* is configured with Algorithm 2.

Spend verifies whether the face value of the notes used matches the face value of the newly generated notes after transfer and whether the type of notes matches. It also verifies ownership of the notes. After the *Spend* process, oh, $nh1$, $nh2$, $enn1$, and $enn2$ are recorded. Also, state of on is recorded as $SPENT$, and the state of $n1$ and $n2$ is recorded as $VALID$.

Spend, unlike $Create$, does not expose any information about face value or type of token in the process. This allows the user to transfer value while ensuring privacy.

3.2.4 Liquidate

Liquidate is the process of converting notes back to tokens and is configured as Algorithm 3. *Liquidate* is similar to $Create$, except for further verification of ownership. The state of n is recorded as $SPENT$ after *Liquidate*. In addition, as with $Create$, the amount and type of token included in the note are exposed when *Liquidate* occurs.

Algorithm 2: Spend

Input:
- private
  on: old note to be spent, $oldNote$
  $nn1$: new note 1 to be created, $newNote1$
  $nn2$: new note 2 to be created, $newNote2$
  sk: secret key of $oldNote$
- public
  oh: hash of on
  $nh1$: hash of $nn1$
  $nh2$: hash of $nn2$

Output:
p: zero-knowledge proof

Function `SpendProve`**:**
- $oh = hash(on)$
- $nh1 = hash(nn1)$
- $nh2 = hash(nn2)$
  `// check note hash with note data`
- $on_{pk} = proveOwnership(sk)$
  `// prove ownership of note`
- $on_v = nn1_v + nn2_v$
- $on_{type} = nn1_{type} = nn2_{type}$
  `// check value and type`
- **return** p

Input:
oh: hash of on
$nh1$: hash of $nn1$
$nh2$: hash of $nn2$
$enn1$: encrypted $nn1$
$enn2$: encrypted $nn2$

Output:
s: bool type value indicating result of verification

Function `SpendVerify`**:**
- $s = verifyProof(oh, nh1, nh2, enn1, enn2)$
- **return** s

Algorithm 3: Liquidate

Input:

- private

n: note to be liquidated

sk: secret key of $oldNote$

- public

nh: hash of n

v: value of n

t: token type of n

Output:

p: zero-knowledge proof

Function `LiquidateProve`**:**

$nh = hash(n)$ `// check note hash with note data`

$on_{pk} = proveOwnership(sk)$ `// prove ownership of note`

$n_v = v$

$n_{type} = t$ `// check value and type`

return p

Input:

nh: hash of note

v: value of note

t: type of note

p: zero-knowledge proof

Output:

s: bool type value indicating result of verification

Function `LiquidateVerify`**:**

$s = verifyProof(nh, v, t, p)$

return s

3.3 Smart Note

Smart note is a special form of note that can prove ownership through other notes. In smart notes, pk contains hash of other note, not the public key, as follows: To use smart notes, you must prove ownership of other note corresponding to pk.

Definition 3. *Smart Note Smart note is special forms of notes that can prove ownership through other notes.*

$$SmartNote = [noteHash, v, type, vk, salt]$$

The advantage of smart note is that even though they do not know the other party's public key, they can transfer ownership only by using hash of the notes. This will be used to trade without knowing the other party's public key in the subsequent trading process. In addition, the hash of other note in smart notes does not reveal the owner of the smart note. Therefore, third parties cannot gain meaningful information through smart notes.

3.3.1 Convert

$Convert$ is the process of converting smart notes into general notes. Ownership of $smartNote$ is verified through $originNote$, and a new note, $newNote$ will be generated. $Convert$ is configured as Algorithm 4.

The key to $Convert$ is that ownership is proven using the public key of on, not the public key of sn. The $Convert$ process is simply a conversion of smart notes to notes, so the amount of token remains unchanged. After $Convert$, nh and enn are recorded, state of sn is $SPENT$ and nn is recorded as $VALID$.

Algorithm 4: Convert

Input:

- private

sn: note to be converted to new note, $smartNote$

on: note used to prove ownership of sn, $originNote$

nn: new note to be converted from sn, $newNote$

sk: secret key of on

- public

sh: hash of sn

oh: hash of on

nh: hash of nn

Output:

p: zero-knowledge proof

Function `ConvertProve`**:**

$sh = hash(sn)$

$oh = hash(on)$

$nh = hash(nn)$

`// check note hash with note data`

$on_{pk} = proveOwnership(sk)$

`// prove ownership of note`

$sn_v = nn_v$

$sn_{type} = sn_{type}$

`// check value and type`

return p

Input:

sh: hash of sn

oh: hash of on

nh: hash of nn

enn: encrypted nn

p: zero-knowledge proof

Output:

s: bool type value indicating result of verification

Function `ConvertVerify`**:**

$s = verifyProof(sh, oh, nh, enn, p)$

return s

4 Trading

Previously, we covered the form and structure of basic notes, the process of converting tokens into notes, converting notes into tokens, and how to transfer notes. Based on this, this chapter describes a private trading protocol.

4.1 Order

On zk-DEX, the order consists of:

$$Order = [makerNoteHash, makerVk, sourceToken, targetToken, price, takerNoteHash, parentNoteHash]$$

The definition of each item is as follows.

- makerNoteHash: The hash of the makerNote, which is the note owned by the maker.
- makerVk: Viewing key of the maker. Used for encrypting takerNote.
- sourceToken: the token type of the markerNote, the token to which the maker wants to trade with targetToken.
- targetToken: token type of takerNoteToMaker
- price: number of targetToken to exchange with 1 sourceToken
- takerNoteHash: hash of note sent by taker to the maker
- ParentNoteHash: parent note of takerNoteToMaker

Each item in the transaction is registered through $MakeOrder$, $TakeOrder$, to be described with later, and the transaction is liquidated through $SettleOrder$.

4.2 Make Order

To proceed with the transaction, maker, the creator of the transaction, must create the transaction through $MakeOrder$. $MakeOrder$ will register $makerNote$, a note owned by maker, in the order book, and is configured as Algorithm 5.

When the $MakeOrder$ is completed, a new $order$ is created and mh and $sourceToken$, $targetToken$, $makerVk$, and $price$ are registered on the $order$ after verification of being the same type as $sourceToken$ type. Also, the state of mn is recorded as $TRADING$. When registering $order$, privacy for the type of token is not guaranteed because you must record the token pair you want to trade with.

Algorithm 5: MakeOrder

Input:
- private
  mn: maker's note to be listed on order, $makerNote$
  sk: secret key of mn
- public
  mh: hash of mn
  t: type of mn, $sourceToken$

Output:
p: zero-knowledge proof

Function `MakeOrderProve`**:**
  $mh = hash(mn)$
  // check note hash with note data
  $mn_{pk} = proveOwnership(sk)$
  // prove ownership of note
  $mn_{type} = t$
  // check type
  return p

Input:
mh: hash of mn
t: type of mn, $sourceToken$
mvk: maker's viewing key, $makerVk$
$price$: price of $sourceToken$ in $targetToken$
p: zero-knowledge proof

Output:
s: bool type value indicating result of verification

Function `MakeOrderVerify`**:**
  $s = verifyProof(mh, t, mvk, price, p)$
  return s

4.3 Take Order

$TakeOrder$ is the process that takes the order made through $MakeOrder$. With $TakeOrder$, taker will use his own $parentNote$ to conditionally transfer $takerNote$ to maker. $TakeOrder$ is configured as Algorithm 6.

When $TakeOrder$ is complete, th and ph, which is proof of type verification of $order_{targetToken}$, are registered on target $order$. Through this process, taker will transfer ownership of tn using a smart note to maker. tn_{pk} is $order_{makerNoteHash}$, which allows the owner of that note hash to use tn. At the time $TakeOrder$ is complete, the state of tn is recorded as $TRADING$, so maker cannot use tn without settlement, the final step of trading, and the state of pn is recorded as $SPENT$.

4.4 Settle Order

When $TakeOrder$ is complete, the creator of that $order$ must settle the order through $SettleOrder$. Using $makerNote$ and $takerNote$ registered on $order$, maker can settle under $order_{price}$, creating $rewardNote$, $paymentNote$, and $changeNote$. $rewardNote$ is a $order_{sourceToken}$ type note given to the taker, and $paymentNote$ is of the $order_{targetToken}$ type note paid to the maker. $changeNote$ is a note that returns change, and the owner is determined by v of $makerNote$ and $takerNote$. $SettleOrder$ is configured as Algorithm 7.

There are several verification processes on $SettleOrder$, but the key is to calculate the correct exchange rate using the price of the token registered by maker and taker. If the taker takes more than the amount registered by the maker, the change will be returned to taker after settlement, otherwise change will be given to the maker. When $SettleOrder$ is complete, the order is settled and rh, ph, ch is recorded.

Algorithm 6: TakeOrder

Input:

- private

pn: taker's note used to create tn, $parentNote$

tn: taker's note to be listed on order, $takerNote$

sk: secret key of pn

- public

ph: hash of pn

th: hash of tn

etn: encrypted tn

$order$: order to be taken

Output:

p: zero-knowledge proof

Function `TakeOrderProve`**:**

$ph = hash(pn)$

$th = hash(tn)$

// check note hash with note data

$pn_{pk} = proveOwnership(sk)$

// prove ownership of note

$tn_{pk} = order_{makerNoteHash}$

// check if it is sent to maker

$pn_{type} = tn_{type} = order_{targetToken}$

// check type

$etn = encrypt(order_{makerVk}, tn)$

// check it is encrypted with maker's viewing key

return p

Input:

ph: hash of pn

th: hash of tn

etn: encrypted tn

$order$: order to be taken

p: zero-knowledge proof

Output:

s: bool type value indicating result of verification

Function `TakeOrderVerify`**:**

$s = verifyProof(ph, th, etn, order, p)$

return s

Algorithm 7: SettleOrder

Input:
- private

mn: maker's note listed on $order$, $makerNote$
tn: taker's note listed on $order$, $takerNote$
rn: note to be given to taker, $rewardNote$
pn: note to be given to maker, $paymentNote$
cn: change note, $changeNote$
sk: secret key of mn

- public

mh: hash of mn, th: hash of tn, rh: hash of rn, ph: hash of pn, ch: hash of cn
ern: encrypted rn, ecn: encrypted cn
$order$: order to be settled

Output:
p: zero-knowledge proof

Function `SettleOrderProve`**:**
- $mh = hash(mn)$, $th = hash(tn)$, $rh = hash(rn)$, $ph = hash(pn)$, $ch = hash(cn)$
- $mn_{pk} = proveOwnership(sk)$
- $tn_{pk} = mh$
- $rn_{pk} = order_{parentNoteHash}$
- **if** $mn_v >= tn_v / order_{price}$ **then**
  - $rn_v = tn_v / order_{price}$
  - $pn_v = tn_v$
  - $cn_v = mn_v - tn_v / order_{price}$
  - $cn_{pk} = mh$
  - $rn_{type} = mn_{type} = cn_{type}$, $pn_{type} = tn_{type}$
- **else**
  - $rn_v = mn_v$
  - $pn_v = mn_v * order_{price}$
  - $cn_v = tn_v - mn_v * order_{price}$
  - $cn_{pk} = order_{parentNoteHash}$
  - $rn_{type} = mn_{type}$, $pn_{type} = tn_{type} = cn_{type}$

`// check every note is settled correctly`

- $ern = encrypt(tn_{vk}, rn)$
- **if** $mn_v < tn_v / order_{price}$ **then**
  - $ecn = encrypt(tn_{vk}, cn)$

`// check every note is encrypted correctly`

- **return** p

Input:
mh: hash of mn, th: hash of tn, rh: hash of rn, ph: hash of pn, ch: hash of cn
ern: encrypted rn, ecn: encrypted cn
$order$: order to be settled
p: zero-knowledge proof

Output:
s: bool type value indicating result of verification

Function `SettleOrderVerify`**:**
- $s = verifyProof(mh, th, rh, ph, ch, ern, ecn, order, p)$
- **return** s

4.4.1 Necessity of Settlement

The issue will be revisited in 5, but the settlement process of zk-DEX clearly has its drawbacks in that the maker has to settle the transaction manually. Nevertheless, this additional settlement process is essential to ensure privacy in the trading process. This is because no encryption algorithm enables communication of information about the encrypted note owned by maker to taker without this settlement process.

Let's look into the issue in detail. To eliminate the settlement process, orders must be automatically settled as soon as the taker takes it. However, to ensure privacy, trading volume of the order should not be revealed to the exchange system as well, so we cannot rely on the exchange to settle the orders automatically. Therefore, in this case, the taker must be able to settle it on its own.

Settlement refers to the process of sending and receiving tokens in accordance with the token price and amount. That means that for settlement, the maker and taker must be aware of the exact amount of the order in advance. However, since the amount of tokens registered by the maker cannot be identified at the time of the order being created, it is impossible for the taker to settle the order.

Of course, we cannot figure out the amount of tokens the maker registered, but there are ways to calculate them. A typical example is to use multikey-homomorphic-encryption[11]. Multikey-homomorphic-encryption allows the results of each user's operation on each encrypted value to remain the same as those of the original value. Let's define the homomorphic encryption function as H, say mv for the amount maker wants to trade, and tv for the amount taker wants to trade. Maker will be able to register the same encrypted value as $H(mv)$, and taker will also be able to register $H(tv)$ to calculate $H(mv * tv)$ with the price.

The problem, however, is not whether the calculation is possible, but how to decode encrypted values. The encrypted value of $H(mv)$ initially registered by the maker is the value that only the maker can decrypt. Again, the same problem arises because the maker is unable to identify the taker at the time of creating the order. Settlement itself is possible with homomorphic encryption, but the taker will not be able to decrypt the encrypted values after settlement. Therefore, it is also impossible for the taker to settle orders.

4.4.2 Incentive of settlement

Zk-DEX guarantees a fee return for the makers. Traditionally, makers, who provide liquidity to the market, were given privileges from the exchange. zk-DEX needs to provide greater rewards because, in addition to this liquidity provider role, makers also carry out the settlement process. Thus, makers can earn commission revenue rather than paying transaction fees.

4.5 Cancel Order

There are two cases in which an order can be cancelled during the trade. First, when the maker cancels the order before it is taken. Second, if the maker fails to settle after the order is taken, the taker can cancel the order.

4.5.1 Cancel Order Before Taken

The maker may cancel the order before the order is taken. In this case, the state of the $makerNote$ changes from $TRADING$ to $VALID$, which in turn makes the note available for general use.

4.5.2 Cancel Order After Taken

Taker may also cancel the order if the maker does not settle within a certain time frame. In this case, the state of the $takerNote$ changes from $TRADING$ to $SPENT$ and the state of $parentNote$ from $SPENT$ to $VALID$.

5 Limitations

zk-DEX have enabled privacy ensured trading in order-book based DEX, but has some limitations. In this section, we describe the limitations of zk-DEX and potential solutions to mitigate or improve the protocol.

5.1 Additional Settlement

In zk-DEX, manual settlement is added to the trading process. Orders are not automatically settled when the taker takes the order, so the maker should settle the order manually. The taker also has to wait for settlement after taking orders. It means that for orders to be settled immediately, maker should be online at all times. However, because Maker is also a normal user, it is hard to expect such a role. Therefore, it is inevitable that there will be a period of delay in trading, which results in low usability.

5.1.1 Solution

In zk-DEX, makers are given economic incentives to settle orders since they can profit from trading fees as a reward. In return for these benefits, we can enforce a more active role for the maker in the settlement process. For example, makers can earn fees if the order is settled within a given time but will be subject to a certain penalty if it is not. This will economically encourage and enforce quick settlement. However, since this is also an inducement to speed up settlement, it is impossible to prevent malicious delays or problems with network failures.

5.2 Trivial Order

To ensure full privacy, no one can find out about the amounts in orders for the maker. Exploiting this, malicious makers can create several orders using notes containing few tokens of near-zero, to attack honest users. Takers will not be able to use the notes until the order is settled, and even if the order is settled, because the processed volume is close to zero, takers cannot trade during that time.

Taker can also take advantage, by taking orders with notes with few tokens to attack honest makers. In zk-DEX, an order cannot be taken by another taker at the same time. Even if the order

is settled, the trading volume is also close to zero, keeping makers unable to trade until the order is settled.

5.2.1 Solution

One way to address this is to limit the minimum volume of trades. For example, when an order is created, a minimum volume will be set, which both the maker and taker must prove that they trade more than that amount. In this case, both makers and takers will be able to identify the minimum amount that can be traded in advance, thus mitigating the problem above. However, this method reveals partial information about the face value of notes, thus sacrificing some privacy.

6 Further Research

6.1 Privacy Enhancement

Currently, the protocol in zk-DEX does not reveal the identity of the owner and the face value of the note, but information about input notes are public. Therefore, it is possible to track record of notes from deposit to withdrawal.

Merkle tree and unlinkable tags used in Hopper and Ethereum 934 may be implemented to solve this issue. All note hash may be added to the merkle tree, and unlinkable tag of that hash is also recorded. Before using notes, it should be verified through zero-knowledge proof that hash of notes are included in the merkle tree and that the unlinkable tags have not been used. This makes it impossible to track the history of notes in zk-DEX.

6.2 Brokerage

A broker can be assigned to improve the usability of zk-DEX. Users will send notes to a broker, along with the requirements of the order. The broker may match these orders or choose to trade with another broker. The broker can identify all trading amounts in advance from the time receiving the notes, so the order can be settled without settlement.

The important thing is that these processes can be done securely, even if they do not trust brokers. Users will be able to register their own note hash in the smart contract when using brokerage services and take ownership of the note hash of smart notes that are created after settlement. In other words, the broker does not take custody; they simply match orders.

Of course, privacy will not be fully guaranteed from the moment you use a broker. This is because the broker must be aware of all the information in each note in advance.

6.3 Delegated Proving

In zk-DEX, all trading processes require generating zero-knowledge proof, so if users use a device with low computational power, the time it takes to generate proof can be very long, making the protocol unavailable for use.

One possible solution is to delegate the computation work of generating proof to a third party. Users who cannot generate proof can delegate the operation by passing on necessary inputs to others with sufficient computational power. In this case, the delegatee will be able to identify the identity of the trader and the volume because all the inputs required will be transferred.

Therefore, it is necessary to distinguish the scope of delegated operations as addressed in ZEXE. ZEXE pointed out that delegating all of the computation work to a transaction would also delegate the user's transaction signature, which could expose the secret key. To address this issue, only operations that do not need signatures are delegated to a third-party, and the user should generate proof of the signature merging them through recursive zero-knowledge proof[**recursive-zkp**].

In a similar way in zk-DEX, it is possible to separate proof computations and delegate the operation without exposing the user's secret key. This enables users to delegate operations efficiently while ensuring a maximum level of privacy.

6.4 Interoperation with other DEXs

zk-DEX can also interoperate with other DEX ecosystems. In the process, zk-DEX serves as a dark pool that can provide anonymous liquidity to other DEXs and share liquidity with other DEXs at the same time.

This can be done by having relayers in zk-DEX, a liquidity provider that connects with external DEXs. Relayers will receive notes and order conditions from the user wishing to trade, all encrypted and recorded in the smart contract. If a relayer has delivered a note to a user that meets the conditions, it will earn ownership of the note delivered by the user and take trading fees. Relayers must trade orders separately in an external DEX, and in this process, user notes cannot be converted to original token to ensure privacy. Therefore, information about the notes used in relaying can only be obtained from the owner and relayer. Also note that all of these processes can be done through smart contacts, without trusting a third-party.

7 Conclusion

zk-DEX enables DEX to support full privacy through two rounds of conditional transactions using zero-knowledge proof. Also, a separate encrypted communication channel is not needed for traders to agree on orders in advance. Instead, orders can be settled through the DEX itself. It takes additional time for orders to be settled, since the settlement of the order must be carried out by the user, not by the exchange system. There is also a possibility of malicious attacks, registering orders that contain very low volume of orders to hinder other trades. These are still the inherent problems of zk-DEX, even though there are solutions to mitigate it. Nevertheless, We think zk-DEX can certainly help users because it makes privacy-guaranteed trading more convenient.

References

[1] *A Next-Generation Smart Contract and Decentralized Application Platform*. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[2] Arpit Agarwal. *ZkDai Design Doc*. URL: https://docs.google.com/document/d/1z3ZRLLD-wvgERe_KO5VhqxEJDqiBd3xxfLO1pYk7Z0k/edit.

[3] argentlabs. *Introducing Hopper: Mobile web-friendly privacy for Ethereum*. URL: https://medium.com/argenthq/introducing-hopper-mobile-web-friendly-privacy-for-ethereum-d02a8c400dad.

[4] *Cryptocurrency market cap*. URL: https://coinmarketcap.com/.

[5] *Decentralized exchange*. URL: https://en.bitcoinwiki.org/wiki/DEXes.

[6] *Decentralized exchange trading volume*. URL: https://dex.watch/.

[7] *DECENTRALIZED VS CENTRALIZED EXCHANGES: ADVANTAGES AND DISADVANTAGES*. URL: https://hacken.io/research/education/decentralized-and-centralized-exchanges-advantages-vs-disadvantages-aa9a27da4584/.

[8] Tom Elvis Jedusor. *MimbleWimble Origin*. URL: https://github.com/mimblewimble/docs/wiki/MimbleWimble-Origin.

[9] Brandon Kite. *zk-dex*. URL: https://github.com/Voxelot/zk-dex.

[10] Wanseob Lim. *Ethereum 9¾: Send ERC20 privately using Mimblewimble and zk-SNARKs*. URL: https://ethresear.ch/t/ethereum-9-send-erc20-privately-using-mimblewimble-and-zk-snarks/6217.

[11] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan. *On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption*. URL: https://eprint.iacr.org/2013/094.

[12] *Merkle Mountain Ranges*. URL: https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

[13] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/bitcoin.pdf.

[14] *Ring signature*. URL: https://en.wikipedia.org/wiki/Ring_signature.

[15] Amir Herzberg Ryan Henry and Aniket Kate. *Blockchain Access Privacy: Challenges and Directions*. URL: https://www.researchgate.net/profile/Amir_Herzberg/publication/326855148_Blockchain_Access_Privacy_Challenges_and_Directions/links/5b80adb292851c1e12304c1Blockchain-Access-Privacy-Challenges-and-Directions.pdf.

[16] *Stealth address*. URL: https://monero.stackexchange.com/questions/1500/what-is-a-stealth-address/1506#1506.

[17] Kendrick Tan. *Introducing Heiswap - An Ethereum Mixer*. URL: https://kndrck.co/posts/introducing_heiswap/.

[18] Patrick P. Tsang and Victor K. Wei. *Short Linkable Ring Signatures for E-voting, E-cash and Attestation*. URL: https://eprint.iacr.org/2004/281.pdf.

[19] Dr Zachary J. Williamson. *The Aztec Protocol*. URL: https://github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf.

[20] *Zero-knowledge proof*. URL: https://en.wikipedia.org/wiki/Zero-knowledge_proof.