Proposing a design for a smart contract $W$

which provides these functionalities

- $W$

is a smart contract which shields txn.

- There is only one smart contract ($W$

) for all addresses.

- deposit any

amount of ETH in $W$

- transfer any

amount of ETH to another address. Here, the eth remains in $W$

, just the internal accounting changes so that the transferred ETH is now controlled by the recipient.

- withdraw any

amount of ETH from $W$

to an address.

The design can easily be extended to shield ERC20, 721 and 1155 transfers as explained at the end.

# Setting up the stage

$A=aG$

and $B=bG$

are ECC public keys, $G$

is the generator, $a$

and $b$

are private keys.

# $A$

wants to deposit $x$

eth to $W$

.

$A$

generates a random secret $s$

and sends $H(sA,x)$

as commitment (leaf in a merkle tree), along with a proof and $x$

eth.

The proof proves that —

- $H(sA,x)$

is a hash of $(saG, msg.value)$

. $G$, $msg.value$

are public values. Note that this can only be proved by someone knowing the private key to $A$

.

H(sA,x)

is added to the merkle tree as a leaf. We call this an unspent commitment as this deposit has been not been transferred or withdrawn.

# A

wants to transfer y

eth to B

All the transfers happen inside the contract, so W.balance

doesn't change and using cryptography and ZK, observers aren't able to recognize the amount being transferred, and the addresses between which the transfer is happening.

A

wants to transfer y

eth to B

. A

generates another random secret r

, and computes B_r = rB

. It nows sends a proof to W

along with public inputs H(sA,x,1)

(nullifier), H(sA,x-y), H(B_r, y)

.

The proof proves that —

- H(sA,x)

is a hash of (saG,x)

, and it has been committed (using a merkle proof).

- H(sA,x,1)

is a hash of (saG,x,1)

.

- H(sA,x-y)

is a hash of (saG,x-y)

.

- H(B_r, y)

is a hash of (B_r,y)

.

W

maintains a boolean hash-map m

keyed on hash values. It first checks if m[H(sA,x,1)]

is true. If so, it rejects the transaction. Otherwise, it sets it to true, and then proceeds. This is done to prevent double spending. The commitment H(sA,x) is now a spent commitment since it can’t be used anymore.

Now, it adds H(sA,x-y), H(B_r,y)

as commitments (leaves in the merkle tree). Again, these are unspent commitments.

At the same time, A

shares r,y

with B

privately off-chain or by using (EC)Diffie-Hellman. A

can alternatively announce r

publicly.

A

can claim to transfer y

eth to rB

, but the protocol doesn’t enforce it, so A

can send it to another secret. Similarly, B

can claim that no eth was tranferred to it.

Since H(B_r,y)

is visible on-chain, this conflict can be resolved. Both parties can compute this hash value to prove their claim or to refute other party’s claim.

# B

wants to withdraw from W

Assume B

knows the secret s

, and it has x

eth associated with it. B

wants to withdraw y

eth from this secret. It now sends a proof to W

along with public inputs H(sG, x, 1), H(sG,x-y)

along with public value y

.

The proof proves that —

- H(sG,x)

is a hash of (sG,x)

, and it has been committed (using a merkle path).

- H(sG,x,1)

is a hash of (sG,x,1)

.

- H(sG,x-y)

is a hash of $(sG,x-y)$

.

$W$

first checks if $H(sG,x,1)$

is already used by checking $m$

. If not, then it first sets this hash to true in $m$

.

Now, it adds $H(sG,x-y)$

as commitment, and then transfers $x$

eth to $B$

.

## Merging two secrets

It can be a pain to have your eth split across different secrets. To withdraw eth from $W$

, you can only withdraw an amount included in just one secret. Hence, it would be nice if you can combine all your eth under one secret. Here's the mechanism.

Suppose $A$

has two commitments corresponding to $(s_1G,x)$

and $(s_2G,y)$

, and you want to combine these eth amounts ($x$

and $y$

) to one secret $(s_1G,x+y)$

.

$A$

provides a proof to $W$

along with public inputs $H(s_1G,x,1), H(s_2G,y,1), H(s_1G,x+y)$

The proof proves that —

- $H(s_1G,x)$

is a hash of $(s_1G,x)$

.

- $H(s_1G,x,1)$

is a hash of $(s_1G,x,1)$

.

- $H(s_2G,y)$

is a hash of $(s_2G,y)$

.

- $H(s_2G,y,1)$

is a hash of $(s_2G,y,1)$

.

- $H(s_1G,x+y)$

is a hash of $(s_1G, x+y)$

.

As above, $W$

first checks with $m$

, the values for the nullifiers values. If they are false, then it sets them to true, then sets $H(s_1G,x+y)$

as a commitment. $s_2$

can be discarded now.

## Viewing keys

- A commitment in $W$

is of the form $H(sC,x)$

where $s$

is a secret, $C=cG$

is an ECC public key and $x$

is the amount of ETH associated with it.

- Knowing $s,c,x$

means controlling this ETH. You can transfer or withdraw it.

- Knowing $s$, $C$, $x$

means you can verify the hash commitment.

- If you want to disclose a transfer which involves your address $C$

, you can disclose the secret $s$

and the amount $x$

. This knowledge won't give anyone the control of ETH, only the ability to verify that you sent or received this amount.

- If you want to reveal your balance in $W$

to someone, you can disclose the secret $s$

and amount $x$

for all your commitments.

## Enable ERC20, 721 and 1155 transfers

- ERC20: Use $H(sA, ERC20\_addr, amount)$

as commitments.

- ERC721: Use $H(sA, ERC721\_addr, tokenId)$

as commitments.

- ERC1155: Use $H(sA, ERC1155\_addr, tokenId, amount)$

as commitments.

If we want all of this in the same contract, then a generalized commitment will be of the form $H(sA, addr, tokenId, amount)$

. We can use zero

where a field is not required.

## Open Questions

- Is it possible to fill up the merkle tree blocking further interaction with the protocol?