Confidential Wrapped Ethereum

Artem Chystiakov, Mariia Zhvanko
Distributed Lab

June 2025

1 Introduction

Transparency is one of the key benefits of public blockchains. However, the public visibility of transactions potentially compromises users' privacy. The fundamental challenge is to balance the intrinsic benefits of blockchain openness with the vital need for individual confidentiality [1].

The goal is to create a permissionless, public good protocol that will obfuscate users' financial activity within ETH tokens by encrypting their balances and transfer amounts. This will allow for confidential peer-to-peer payments, donations, and acquisitions in ETH without relying on centralized entities.

The proposal suggests creating a confidential version of wrapped Ethereum (cWETH) fully within the application layer. The solution combines the Elliptic Curve (EC) Twisted ElGamal-based commitment scheme to preserve confidentiality and the EC Diffie-Hellman (DH) protocol to introduce accessibility limited by the commitment scheme. To enforce the correct generation of commitments, encryption, and decryption, zk-SNARKs are utilized.

1.1 Difference from the existing protocols

There are at least two known solutions that do not require the protocol layer modifications and can be implemented as is.

The Solana [2] and Zether [3] approaches may seem very similar at first glance, as they also use ElGamal commitments. However, the main distinguishing factor is the need to solve the discrete logarithm problem to access the encrypted balance.

Solana partially mitigates this by maintaining a separate decryptable balance. But the main difference is that the encryption scheme used doesn't support aggregation, and such a balance works mainly as a cache, which is updated when the pending amount is moved to the actual one. However, it is still required to decrypt the balance represented as the ElGamal commitment, which reintroduces the necessity to solve the discrete logarithm problem. Although this process can be simplified by dividing the value into chunks, the cWETH protocol proposes an encryption scheme that is aggregateable and is managed

along with the ElGamal commitment to avoid computing the discrete logarithm at all.

Another difference lies in the ZK proofs used by cWETH. Both Solana and Zether approaches rely on the Sigma protocols and bulletproofs, while this proposal is based on the zk-SNARKs. The tradeoff here is the need for a trusted setup, but this can be mitigated by using existing trusted setups, such as Plonk's universal setup, which has proven to be secure over time.

2 cWETH Overview

2.1 Wrapping ETH

The process of setting up the confidential account is considered to be a part of the ETH wrapping operation. Along with the usual wrap logic, upon the first deposit to the cWETH contract, each user provides a babyJubJub public key that will be used for the management of balances.

This public key is employed in the commitment and encryption of token amounts and balance computations. The key pair generation and initial deposit flow is depicted in the following diagram:

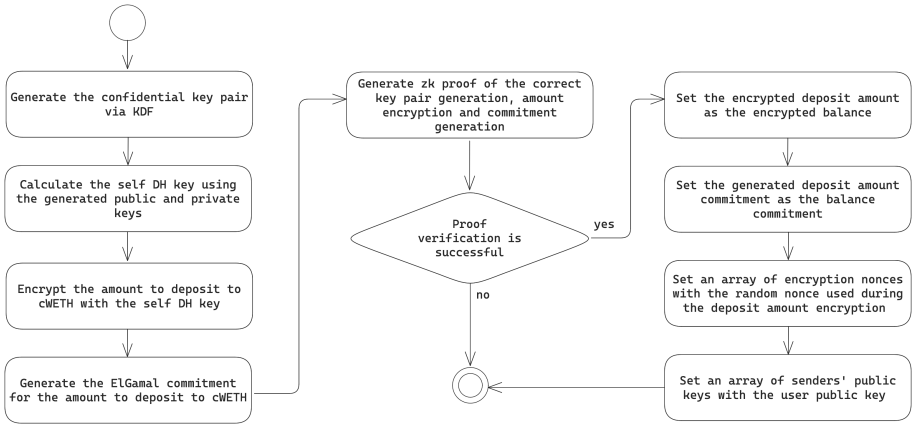

Figure 1: Initial deposit to cWETH flow

After the first deposit to the cWETH contract, initial balances are updated with the values calculated according to the same formulas used for a confidential transfer, considering that the user is both the sender and the receiver. Technical details related to these values and the key pair generation process are provided in further sections of this proposal.

The unwrapping operation is very similar conceptually, hence it will not be covered in this section.

2.2 Balance management

To avoid computing the discrete logarithm to calculate the balances hidden with ElGamal, two parallel representations of balances are maintained:

- *ElGamal commitment* that is only needed for ZK proofs verification of the balance knowledge.
- *Balance encrypted with the DH shared key* which enables users to decrypt their actual balance data. Access to the senders' public keys array and the encryption nonces array is required to perform the decryption.

Because ZK proofs are generated from the user's current balance, changing it before the transaction execution may invalidate the proof. To address this issue, both types of balance (ElGamal commitment and DH-encrypted) have to be presented in two separate states:

- Pending balance to receive tokens.
- Actual balance to spend tokens.

Users can move tokens from their pending balance to the actual one at any time. Additionally, it could be done at the end of each transfer initiated by the user. This approach will consolidate the balance update into a single operation, reducing the gas usage compared to two separate transactions.

2.3 Confidential transfer

To initiate a confidential transfer, it is necessary to provide the token amount represented in four different forms:

- Hidden inside the receiver's public key-based ElGamal commitment.
- Hidden inside the sender's public key-based ElGamal commitment.
- Encrypted with the receiver's public key-based DH shared key for incrementing the receiver's balance.
- Encrypted with the sender's public key-based DH shared key for the new sender's balance.

The diagram below illustrates the confidential transfer process:

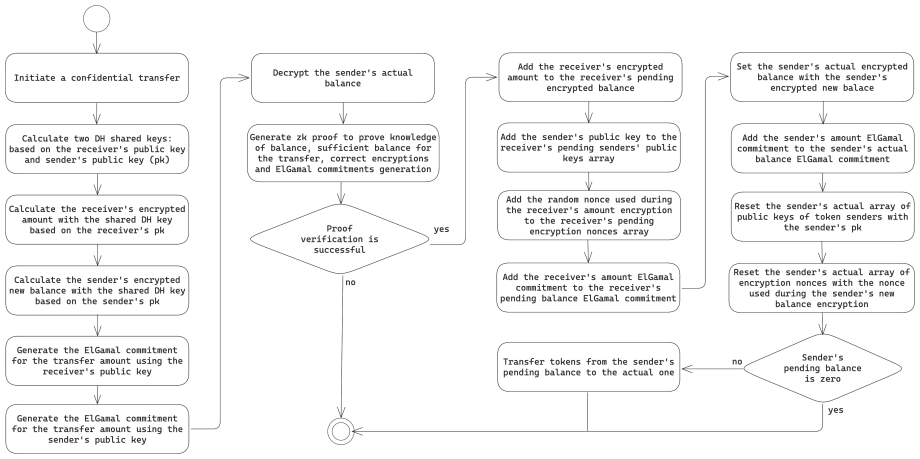

Figure 2: Confidential transfer flow

During the transfer, both types of balances and their associated amounts are updated to ensure consistency between different balance representations.

Note that there exist two separate versions of the same values for both encrypted with the DH shared key and ElGamal commitment representations: the sender's public key-based and the receiver's public key-based. This is caused by different approaches used during the encryption and commitment generation, technical details of which are provided in the further sections of the proposal.

3 Confidentiality Math

3.1 KDF for confidential key pair

The protocol utilizes the deterministic KDF approach to generate a confidential key pair. The user has to sign an EIP-712 structured message with their Ethereum (ECDSA) private key. The hash of the signature is the babyJubJub private key.

The EIP-712 message is obtained as follows:

```
bytes32 KDF_MSG_TYPEHASH = keccak256("KDF(address cWETHAddress)");

bytes32 kdfStructHash = keccak256(
    abi.encode(KDF_MSG_TYPEHASH, cWETHAddress)
);
```

The babyJubJub private key is obtained as follows:

```
signature = eth_signTypedData_v4(kdfStructHash);
privateKey = keccak256(keccak256(signature));
```

Note that it is crucial to never reveal the signature as the private key is directly derived from it.

The key pair must satisfy the following:

$$P = sk \times G,$$

where P — public key,
sk — private key,
G — base point on the elliptic curve.

3.2 EC ElGamal-based commitment scheme

The commitment based on the EC ElGamal scheme allows for a succinct representation suitable for ZK proofs. It also facilitates seamless balance management via its additive homomorphic properties.

The commitment of the balance consists of two parts and is constructed according to the formula:

$$C = b \times H + r \times G$$

$$D = r \times P$$

where H, G — base points on the elliptic curve,
b — user balance,
r — random nonce,
P — user's public key.

To efficiently prove the commitment of the balance through a ZK proof, the user must use the private key to compensate for the lack of knowledge of the aggregated random nonce:

$$C = b \times H + sk^{-1} \times D = b \times H + r \times G$$

The commitment of the transfer amount is calculated differently for the sender and the receiver. The receiver's amount commitment is derived as follows:

$$C_{a_r} = a \times H + r \times G$$

$$D_{a_r} = r \times P_r$$

where P_r — receiver's public key.

The transfer amount commitment used in the aggregation of the sender's balance commitment is calculated as:

$$C_{a_s} = r \times G - a \times H$$

$$D_{a_s} = r \times P_s$$

where P_s — sender's public key.

The balance commitment is computed additively as outlined below:

$$C = \sum_{i=1}^{N} C_{a_i} = \sum_{i=1}^{N} a_i \times H + \sum_{i=1}^{N} r_i \times G$$

$$D = \sum_{i=1}^{N} D_{a_i} = \sum_{i=1}^{N} r_i \times P$$

where N — number of token transfers the user received,
C_{a_i}, D_{a_i} — transfer amount commitment,
a_i — amount transferred to the user,
r_i — random nonce used in the commitment.

3.3 Encryption with the EC DH shared key

Since ElGamal commitments alone do not provide a convenient means for users to access decrypted balances, requiring solving the discrete logarithm problem, an additional form of amount hiding through encryption is employed using the EC DH shared keys.

The DH shared key is derived as follows:

$$K = sk_r \times P_s = sk_s \times P_r = sk_r \times sk_s \times G$$

where sk_s, sk_r — private keys of the sender and receiver, respectively,
P_s, P_r — public keys of the sender and receiver, respectively.

The transfer amount DH-based encryption is performed differently for the sender and the receiver.

The encryption of the transfer amount used to aggregate the receiver's balance is calculated according to the formula:

$$A_r = a + K_x + poseidon(K_x||n) \mod p =$$
$$= a + (sk_s \times P_r)_x + poseidon((sk_s \times P_r)_x||n) \mod p$$

where a — amount to be transferred,
K_x — x coordinate of the DH shared key,
n — random nonce,
p — large prime number.

The random nonce usage is required to solve the problem of the potential transfer data leaks caused by the lack of randomness in the encryption scheme, which can be particularly noticeable in recurring payments. These nonce values

have to be stored together with the senders' public keys so that users can decrypt their balances.

Encrypted amounts are further used to aggregate the encrypted receiver's balance:

$$B_r = \sum_{i=1}^{N} A_i \mod p =$$

$$= \sum_{i=1}^{N} a_i + \sum_{i=1}^{N} (sk_r \times P_{s_i})_x + \sum_{i=1}^{N} poseidon((sk_r \times P_{s_i})_x || n_i) \mod p$$

where A_i — amount encrypted with the DH shared key sent to the user.

According to the formula, the receiver can decrypt their balance as follows:

$$b_r = B_r - \sum_{i=1}^{N} (sk_r \times P_{s_i})_x \mod p - \sum_{i=1}^{N} poseidon((sk_r \times P_{s_i})_x || n_i) \mod p$$

On the sender side, using this approach for encrypting the transfer amounts could potentially lead to managing an infinite number of public keys and random nonces necessary for the balance decryption. To resolve this, each time a user transfers tokens, the lists of existing senders' public keys and random nonces are reset. This is accomplished by calculating the new encrypted sender's balance, as outlined below:

$$B_{s_{new}} = b_s - a + (sk \times P)_x + poseidon((sk \times P)_x || n) \mod p$$

where n — random nonce used in the new balance encryption,
b_s — sender's balance,
a — amount to be transferred,
sk — sender's private key,
P — sender's public key.

After updating the encrypted balance, the sender needs to manage only their own public key and random nonce as a single entry to decrypt the total amount of tokens owned.

The identical logic for the new balance calculation can be used to unwrap cWETH tokens.

4 Solidity Smart Contracts

4.1 State variables

4.1.1 Public keys

First of all, each user is associated with the corresponding public key generated during the first deposit to the cWETH contract. This is managed in a simple

$mapping(address => uint256[2])$.

4.1.2 Balance

Balance of each user is represented in the ElGamal commitment form:

```
struct Commitment {
    uint256[2] C;
    uint256[2] D;
}
```

To decrypt the balance encrypted with the DH shared key, along with the DH balance itself, an array of public keys of the senders and an array of encryption nonces need to be stored:

```
struct DHBalance {
    uint256 encryptedBalance;
    uint256[][2] sendersPublicKeys;
    uint256[] encryptionNonces;
}
```

Because both balance types have to be provided in the form of the pending and actual balances, the final balance storage looks like this:

```
struct Balance {
    Commitment elGamalCommitmentPending;
    Commitment elGamalCommitmentActual;
    DHBalance dhEncryptedPending;
    DHBalance dhEncryptedActual;
}

mapping(uint256 => Balance) internal balances;
```

Note that internal balance accounting is performed using the x-coordinate of the user's public key as the key in the balances mapping.

4.2 Functions

4.2.1 Depositing to cWETH

For the correct confidential account creation, the deposit function is to be provided:

```
function deposit(
    uint256[2] publicKey,
    bytes calldata amountCommitmentData,
    bytes calldata balanceEncryptionData,
    bytes calldata proofData
```

```
) public payable;
```

Alongside a regular ZK proof, the proofData is required to make sure that the user actually owns the private key for the provided public key.

The amountCommitmentData is further decoded as follows:

```
(uint256[2] commitmentC, uint256[2] commitmentD) =
    abi.decode(amountCommitmentData, (uint256[2], uint256[2]));
```

The balanceEncryptionData is decoded as follows:

```
(uint256 encryptedBalance, uint256 encryptionNonce) =
    abi.decode(balanceEncryptionData, (uint256, uint256));
```

4.2.2 Confidential transfer

Having a clear understanding of the commitment and encryption schemes, the transfer function can be specified as follows:

```
function transfer(
    address receiver,
    bytes calldata amountCommitmentData,
    bytes calldata amountEncryptionData,
    bytes calldata proofData
) external;
```

The amountCommitmentData is further decoded as follows:

```
(
  uint256[2] senderCommitmentC,
  uint256[2] senderCommitmentD,
  uint256[2] receiverCommitmentC,
  uint256[2] receiverCommitmentD
) = abi.decode(
    amountCommitmentData,
    (uint256[2], uint256[2], uint256[2], uint256[2])
  );
```

The amountEncryptionData is decoded as follows:

```
(
  uint256 newEncryptedBalance,
  uint256 senderEncryptionNonce,
  uint256 receiverEncryptedAmount,
  uint256 receiverEncryptionNonce
) = abi.decode(
    amountEncryptionData,
```

```
    (uint256, uint256, uint256, uint256, uint256)
  );
```

4.2.3 Withdrawing cWETH

For unwrapping cWETH tokens, the withdraw function is to be provided:

```
function withdraw(
    uint256 amount,
    bytes calldata amountCommitmentData,
    bytes calldata balanceEncryptionData,
    bytes calldata proofData
) external;
```

The amountCommitmentData is further decoded as follows:

```
(uint256[2] commitmentC, uint256[2] commitmentD) =
    abi.decode(amountCommitmentData, (uint256[2], uint256[2]));
```

The balanceEncryptionData is decoded as follows:

```
(uint256 newEncryptedBalance, uint256 encryptionNonce) =
    abi.decode(balanceEncryptionData, (uint256, uint256));
```

5 Circom Circuits

Each parameter described in this proposal is designed to be verifiable on-chain and compatible with zero knowledge circuits.

5.1 Deposit to cWETH circuit

The list of circuit signals for the deposit into the cWETH proof is the following:

Public signals:

- User public key;
- Deposit amount;
- ElGamal commitment of the user's balance;
- ElGamal commitment of the deposit amount;
- User balance after the deposit encrypted with the user's public key-based DH key;

- Encryption random nonce.

Private signals:

- User private key;
- Random nonce used in the ElGamal commitment.

Operating these signals, the circuit must have the following constraints:

1. The provided private key is indeed the private key of the provided public key.
2. The provided deposit amount is proven to be the one committed using the ElGamal-based commitment.
3. The user balance after the deposit is proven to be the one encrypted with the DH shared key.

5.2 Confidential transfer circuit

The list of circuit signals for the confidential transfer proof is the following:

Public signals:

- Sender's public key;
- Receiver's public key;
- ElGamal commitment of the sender's balance;
- ElGamal commitment of the transfer amount based on the sender's public key;
- ElGamal commitment of the transfer amount based on the receiver's public key;
- New sender's balance encrypted with the sender's public key-based DH shared key;
- Random nonce used in the sender's new balance encryption;
- Transfer amount encrypted with the receiver's public key-based DH shared key;
- Random nonce used during the receiver's transfer amount encryption.

Private signals:

- Sender's private key;
- Sender's balance;

- Transfer amount;
- Random nonce used in the sender's ElGamal commitment;
- Random nonce used in the receiver's ElGamal commitment.

Operating these signals, the circuit must have the following constraints:

1. The provided private key is indeed the private key of the provided sender's public key.
2. The provided sender's balance is proven to be the one committed using the ElGamal commitment and to be greater than or equal to the transfer amount.
3. The sender's ElGamal commitment of the transfer amount was generated correctly.
4. The receiver's ElGamal commitment of the transfer amount was generated correctly.
5. The new sender's balance was correctly encrypted with the sender's public key-based DH shared key.
6. The transfer amount was correctly encrypted with the receiver's public key-based DH shared key.

5.3 Withdraw from cWETH circuit

The list of circuit signals for withdrawing cWETH proof is the following:

Public signals:

- User public key;
- Receiver address;
- Withdrawal amount;
- ElGamal commitment of the user's balance;
- ElGamal commitment of the withdrawal amount based on the user's public key;
- Balance after the withdrawal encrypted with the user's public key-based DH shared key;
- Random nonce used during the new balance encryption.

Private signals:

- User private key;

- User balance;
- Random nonce used in the ElGamal commitment.

Operating these signals, the circuit must have the following constraints:

1. The provided private key is indeed the private key of the provided public key.
2. The provided sender's balance is proven to be the one committed using the ElGamal commitment and to be greater than or equal to the withdrawal amount.
3. The ElGamal commitment of the withdrawal amount was generated correctly.
4. The new balance (after the withdrawal) was correctly encrypted with the user's public key-based DH shared key.

References

[1] Pascal Caversaccio. *Ethereum Privacy: The Road to Self-Sovereignty*. 2025. URL: `https://hackmd.io/@pcaversaccio/ethereum-privacy-the-road-to-self-sovereignty`.

[2] Solana Foundation. *Confidential Token Extension*. 2022. URL: `https://spl.solana.com/confidential-token/deep-dive/overview`.

[3] Benedikt Bünz et al. *Zether: Towards Privacy in a Smart Contract World*. 2020. URL: `https://eprint.iacr.org/2019/191.pdf`.