Incognito Solana

17th January 2022

1 Introduction

Incognito Solana implements the **first** zero-knowledge privacy solution for mixing Solana in an **untraceable** manner.

It employs a smart contract that accepts transactions in `Sol` so that the amount can be later withdrawn with no reference to the original transaction. This ensures anonymity for our users, while also providing protection against malicious third parties. Incognito also enforces strict guarantees against double-withdrawals and prevents theft while maintaining user-privacy.

To achieve user privacy, smart contracts are used that accept token deposits from one address and enable their withdrawal from a different address. These smart contracts also work as pools that mix all the deposited assets. When the funds are transferred to a completely new wallet address from such pools, they break the on-chain link between the source and destination of the users' funds, providing asset anonymity guarantees for each user. When a user deposits funds into a pool a private note is generated. The user can use the private note (acting as a private key) to access those funds later.

To ensure additional guarantees of privacy and anonymity, the user can also keep basic rules in mind such as: (a) Allowing a lapse of time between the deposit the withdrawal transaction (b) Mixing their funds extensively within the pool by allowing multiple transactions before recovering their assets.

2 Background and Related Work

CoinShuffle [9], which is one of the earliest (decentralized) mixing protocols, was designed for Bitcoin. It does not require any third party and uses minimal communication overhead for its users. Its main approach includes a group of users who jointly create a single mixing transaction and each of them individually ensure that they won't lose money by performing the transaction. This makes CoinShuffle particularly vulnerable to DoS attacks, where a user joins the mix and then aborts, disrupting the protocol for all other users. Decentralization requires users to interact via a peer-to-peer network to identify each other when performing mix payments. The coordination between users requires communication, and costs grow quadratically. This limits scalability, and the number of users can't exceed 50, thus making it easy for an attacker to create many sybils.

Miximus [5] is a zkSNARK(Zero-Knowledge Succinct Non-Interactive Argument of Knowledge)-based [8] mixer for Ethereum. It uses zkSNARKs to conceal the mapping between depositors and recipients. There are disadvantages of this approach, i.e Miximus only provides anonymity against outsiders – if Alice funds to Bob via Miximus, Alice will know when Bob made the withdrawal transaction. Another crucial limitation of Miximus is that one requires a trusted setup for generating the proving key for the zkSNARK. Once the trusted setup is compromised, the deployer of the contract can steal the funds. This makes Miximus prone to theft.

Mobius[3] is one of the early trustless coin mixers, which was designed for Ethereum. Mobius is a ring-signature-based trust-less coin mixer with minimal on-chain transaction complexity: users of Mobius just need to create a deposit and a withdrawal transaction. However, the gas cost of the withdrawal transaction increases linearly in the number of receivers. This limits the size of possible anonymity sets. Moreover, the number of receivers is capped at 24 due to transaction limits. This might freeze funds in the mixer contract since the gas costs of withdrawal transactions would be greater than the block gas limit. Another tragic problem with Mobius' implementation is that the withdrawal transactions can be front-runned by anyone in the network. This compromises users' privacy, and anyone could steal receivers' funds from the Mobius smart contract. Therefore theft prevention in Mobius' implementation is not guaranteed also.

We design Incognito, which overcomes the above limitations, to provide a seamless and secure protocol for our users on the `Solana` platform. Our protocol builds on the principles proposed by the Tornado Mixing Protocol[7].

3 Cryptographic Primitives

3.1 Elliptic Curves

An elliptic curve is a type of cubic curve whose solutions are confined to a region of space that is topologically equivalent to a torus[1]. The Weierstrass elliptic function $P(z; g_2, g_3)$ describes how to get from this torus to the algebraic form of an elliptic curve. More formally, an elliptic curve over a field K is a non-singular cubic curve in two variables, $f(X, Y) = 0$, with a K-rational point (which may be a point at infinity). The field K is usually taken to be the complex numbers C, reals R, rationals Q, algebraic extensions of Q, $p - adic$ numbers Q_p, or a finite field. An Elliptic curve can also be seen as the set of points described by the following equation.

$$y^2 = x^3 + ax + b$$

where $4a^3 + 27b^2 \neq 0$. The equation above is called Weierstrass normal form for elliptic curves.

A **group** in mathematics is a set for which we have defined a binary operation that we may call *addition* and indicate with the symbol $+$. In order for the set to be a group, *addition* must defined so that it respects the following four properties [1]:

1. **Closure**: If x and y are members of G, then $x + y$ is a member of G
2. **Associativity** : $(x + y) + z = x + (y + z)$
3. **Identity** : There exists an identity element 0 such that $x + 0 = 0 + x = x$
4. **Inverse** : Every element has an inverse, that is for every x there exists y such that $x + y = 0$
5. **Commutativity** : $a + b = b + a$ (In case of an abelian group)

Group Law for Elliptic Curves

We can define a group over elliptic curves. Specifically:

- the **Identity** element is the point at infinity 0
- the elements of the group are the points of an elliptic curve
- The **Inverse** of a point P is the point which is symmetric about the $x - axis$
- **Addition** is followed as: Three points(X, Y, Z) which are aligned and non-zero have 0 as their sum, i.e $X + Y + Z = 0$

3.2 Pedersen Commitments

The Pedersen hash function[6] is a secure hash function that maps a sequence of bits to a compressed point on an elliptic curve (Libert, Mouhartem, and Stehlé).

The Pedersen hash has already been defined in recent literature using Jubjub elliptic curve and 3-bit lookup tables. In this paper, we use a different implementation of the Pedersen hash function using the b-jubjub elliptic curve and 4-bit windows. This hash function requires fewer constraints per bit than using 3-bit windows.

3.3 Elliptic Curve Cryptography

An Elliptic curve is a key-based alternative to encrypting data to standard encryption techniques such as RSA, DSA based on the algebraic structure of elliptic curves over finite fields [1, 2]. ECC derives its robustness from the ECDLP or the Elliptic Curve Discrete Logarithm Problem, which assumes that finding the discrete logarithm of a random elliptic curve element for a publicly known base point is computationally infeasible. The security lies in the inability to compute the multiplicand even if one is shown the original and product points. There are two sets of commonly used algorithms based on ECC i.e 1) Elliptic Curve Diffie Helman Key exchange protocol (ECDH), and 2) Elliptic Curve Digital Signature Algorithm (ECDSA).

3.4 Elliptic Curve Diffie Helman(ECDH)

ECDH or Elliptic Curve Diffie Helman is a key-agreement protocol, and a variant of the Diffie-Hellman algorithm[1] for elliptic curves. It is more than an encryption algorithm. ECDH formalizes how keys should be generated and exchanged between parties. We are free to choose how we encrypt the data using such keys.

Imagine a scenario where two parties (the usual Alice and Bob) want to exchange information securely, so that a third party (the Man In the Middle) may intercept them, but may not decode them. This problem can be solved using the ECDH key agreement protocol.

3.4.1 Protocol

The ECDH Protocol is outlined below:

1. Alice and Bob generate their own private and public keys. Alice generates the public key $H_A = d_A G$ using private key d_A, and Bob generates public key $H_B = d_B G$ using private key d_B. Both Alice and Bob are using the same domain parameters: i.e the same base point G on the same elliptic curve on the same finite field.

2. Alice and Bob exchange their public keys through a communication channel. The attacker(man in the middle) may intercept H_A and H_B as this is an informal channel but when it does, they wouldn't be successful in finding d_A or d_B without solving the discrete logarithm problem.

3. Alice calculates $S = d_A H_B$ (using her private key and Bob's public key), and Bob calculates (using Alice's public key and his private key). S is the same for both Alice and Bob where S can be computed as follows:

$$S = d_A H_B = d_A (d_B G) = d_B (d_A G) = d_B H_A$$

The attacker, however, only knows H_A and H_B and would not be able to find out the shared secret S. This allows seamless exchange of data between Alice and Bob with the protection offered by the protocol.

3.5 Elliptic Curve Digital Signature Algorithm(ECDSA)

3.5.1 Signing Signatures

Alice wants to sign a message with her private key (d_A), and Bob wants to validate the signature using Alice's public key (H_A). Nobody but Alice should be able to produce valid signatures. Anyone should be able to check signatures though. ECDSA works on the hash of the message, rather than on the message itself. One can choose any cryptographically-secure hash function for the same. The hash of the message needs to be truncated to get the bit-length of the has equal to the order of the subgroup i.e n. The truncated hash is denoted by z. The algorithm to sign messages proceeds in the following steps.

- 1. A random integer k is chosen from $1, \cdots, n-1$ (where n is the subgroup order.)

2. We calculate the point $P = kG$ (where G is the base point of the subgroup).

3. We calculate $r = x_P \mod n$ (where x_P is the x coordinate of P). If $r = 0$, then we choose another k and try again.

4. After we get a non-zero r, we $s = k^{-1}(z + rd_A) \mod n$ (k^{-1} is the multiplicative inverse of k). If $s = 0$, we choose another k and try again.

3.5.2 Verifying Signatures

In order to verify the signature Alice needs her public key H_A, the hash z, and the signature (r, s). Verification is done by following the steps:

1. Calculate $u_1 = s^{-1}z \mod n$
2. Calculate $u_2 = s^{-1}r \mod n$
3. Calculate $P = u_1G + u_2H_A$

The signature is valid only if $r = x_p \mod n$.

4 Incognito Protocol

The protocol proceeds as follows:

1. Inserting/depositing money to the smart contract. One can deposit money with the help of a single transaction with a fixed amount (denoted by N) of `Solana`. This is denoted as a coin.

2. Withdrawal of the money from the smart contract can be done in an of the following ways:

   a. N `SOL` is withdrawn with the help of a Relayer with *fee_amount* `SOL` sent as a fee to the Relayer address *RelAddr* and (N - *fee_amount*) to the designated recipient. The *fee_amount* and *RelAddr* is chosen by the sender. The withdraw transaction is initiated by the Relayer and it pays the Gas fee that is supposed to be covered by the *fee_amount*.

   b. The N `SOL` is withdrawn to the designated recipient, the transaction is initiated by the recipient. The recipient should have enough `SOL` to pay Gas fee for the transaction. In that case fee *fee_amount* is considered to be equal to 0.

4.1 Protocol Security

Incognito claims the following security properties:

- **Authenticity:** The protocol doesn't permit withdrawal of coins which haven't been deposited by respective users.
- **Fairness**: The protocol ensures that no coin can be withdrawn twice. Moreover, a coin can only be withdrawn if its parameters (k, r) are known, unless a coin with the same k has been already deposited and withdrawn.
- **Robustness:** If k or r is unknown, a coin can not be withdrawn. If k is unknown to the attacker, he can't prevent someone knows (k, r) from withdrawing the coin. This outline also includes all cases of front-running a transaction.
- **Accountability:** This proof is binding i.e a user can not use the same proof with a different nullifier hash, a new fee amount, or another recipient address.
- **Attack Resistance:** The cryptographic primitives used by Incognito have at least 126-bit security. The security is non-degradable due to its composition.

4.2 Protocol Description

Let $M = \{0, 1\}$. Let e be the pairing operation used in SNARK [8] proofs, which is defined over groups of prime order q. Let

$$H_x : \mathbb{M}^* \to \mathbb{Z}_p$$

where $\mathbb{Z}_p$ be a Pederson hash function . Let

$$H_y : (\mathbb{Z}_p, \mathbb{Z}_p) \to \mathbb{Z}_p$$

be the MiMC [4] hash function. This MiMC hash function operates in the `Feistel`[1] model along with a sponge mode of operation.

We assume a merkle tree M_{tree} of height `20`, where each non-leaf node hashes both of its children with H_y. The leaves of M_{tree} is initialized to zero values. Then, the zero values are gradually replaced with other values from $\mathbb{Z}_p$.
Let us assume that $O(M_{\text{tree}}, i)$ is the Merkle Tree opening for leaf with index i (value of sister nodes on the way from leaf i to the root, which again is denoted by $root$) in the tree M_{tree}.

We use $k \in \mathbb{M}^{248}$ for the `nullifier` and $r \in \mathbb{M}^{248}$ to denote `randomness`. The `Solana` address of the coin recipient is represented by $RecipAddr$.

Let $S[root, hash, RecipAddr, fee\_amount, RelAddr]$ be the following statement of knowledge with values $root, hash, RecipAddr, fee\_amount, RelAddr$ (which are public):

Now,

$S[root, hash, RecipAddr, fee\_amount, RelAddr] = \{k, r \in \mathcal{M}^{248}, i \in \mathcal{M}^{16}, O \in \mathbb{Z}_p^{16}\}$

such that $hash = H_x(k)$ & O is the opening of $H_y(k||r)$ at position l to $root$

$RecipAddr$ and fee_amount are also included in the above context. $hash$ denotes the nullifier hash in this case.

Let $D = (d_p, d_v)$ be the `ZK-SNARK` [8] proof which is used for proving and verifying key pair for S created using a trusted setup procedure.

Let

$$Prove(d_p, M_{\text{tree}}, k, r, i, RecipAddr, fee\_amount, RelAddr) \to P$$

be the proof constructor using d_p and

$$Verify(d_v, P, root, hash, RecipAddr, fee\_amount, RelAddr)$$

be the proof verifier. Let SMC be the smart contract with the following functionality:

- The smart contract stores the last $n = 120$ root values in the history array. For the Merkle tree M_{tree} , it also stores the values of nodes that are on the path from the last leaf to the root. These are values are necessary to compute the next root.
- The smart contract also accepts payments for N `SOL` with data $C \in \mathbb{Z}_p$. The value C is added to the Merkle tree. Then, the path from the last added value and the latest root is recalculated. Finally, the previous root is added to the history array.
- The smart contract verifies the given proof P against the submitted public values $root, hash, RecipAddr, fee\_amount, RelAddr$. After successful verification the contract sends fee fee_amount `SOL` to the Relayer address $RelAddr$ and $(Nfee\_amount)$ `SOL` to address A.
- The contract also verifies whether the coin has been withdrawn before by checking the nullifier hash from the pre-existing proof, and if it hasn't been withdrawn then adds it to the list of nullifier hashes.

4.3 Deposit Mechanism

The mechanism which allows a user to deposit a coin is outlined below.

1. *Configuring seed*: Generate two random numbers $k, r \in \mathbb{M}^{248}$ and compute $C = H_x(k||r)$.
2. *Creating transactions*: Generate a transaction with N `SOL` to contract `C` with data C that should be interpreted as an unsigned 256-bit integer.
3. *Adding Transactions*: If the Merkle tree is not full, then the contract accepts the transaction and adds C to the tree as a new non-zero leaf.

4.4 Withdrawal Mechanism

The mechansim to withdraw a coin (k, r) with position i in the tree is outlined below:

1. *Choose recipients:* We select a recipient address A and fee value $fee\_amount \leq N$;
2. *Compute Opening:* We select the *root* among the stored values in the contract and compute the opening $O(l)$ that ends with the *root*.
3. *Compute Nullifier* : We compute the nullifier $hash = H_x(k)$
4. *Compute Proof*: We instantiate proof P by calling the Prove() function on d_p
5. *Perform Withdrawal*: This can be done in any of the following ways:
   a. A `Solana`-backed transaction is sent to the defined contract `C` with the parameters required i.e *root*, *hash*, *RecipAddr*, fee_amount, *RelAddr*.
   b. One sends a request to the Relayer supplying transaction data *root*, *hash*, *RecipAddr*, fee_amount, *RelAddr*. The Relayer then makes a transaction to contract `C` with the given data.

5 Conclusion

In this paper, we propose Incognito Solana which implements the first zero-knowledge privacy solution for mixing Solana in an untraceable manner. As a decentralized protocol, Incognito smart contracts have been deployed on the Solana Blockchain which makes them immutable – they cannot be changed or tampered by anyone post deployment, even by the initial developers.

As a non-custodial protocol, users keep custody of their crypto-assets while operating within the protocol. With every deposit transaction, they are provided with the private keys that give them direct access to their deposited assets, giving users absolute control over their funds.

The protocol also functions with zk-SNARK [8], which enables zero-knowledge proofs allowing users to demonstrate possession of private keys without needing to reveal it. The code behind Incognito's functioning - smart contracts, circuits, and toolchain will be fully open sourced and auditable by anyone post deployment.

References

[1] Dan Boneh and Victor Shoup. "A Graduate Course in Applied Cryptography". In: (2020).

[2] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Berlin, Heidelberg: Springer-Verlag, 2003. ISBN: 038795273X.

[3] Sarah Meiklejohn and Rebekah Mercer. "Möbius: Trustless tumbling for transaction privacy". In: (2018).

[4] *MiMC.* eprint: `https://byt3bit.github.io/primesym/mimc`.

[5] *Miximus.* eprint: `https://github.com/barryWhiteHat/miximus`.

[6] *Pederson Hash.* eprint: `http://bitly.ws/nMaj`.

[7] Alexey Pertsev, Roman Semenov, and Roman Storm. "Tornado Cash Privacy Solution Version 1.4". In: (2019).

[8] Maksym Petkus. "Why and How zk-SNARK Works". In: *ArXiv* abs/1906.07221 (2019).

[9] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. "Coinshuffle: Practical decentralized coin mixing for bitcoin". In: *European Symposium on Research in Computer Security.* Springer. 2014, pp. 345–364.