Hi community,

I've recently been working on this draft that describes zk-SNARK compatible ERC-721 tokens:

https://github.com/Nerolation/EIP-ERC721-zk-SNARK-Extension

Basically every ERC-721 token gets stored on a Stealth Address that constists of the hash h

of a user's address a

, the token ID tid

and a secret s

of the user, such that stealthAddressBytes

= h(a,tid,s)

. The stealthAddressBytes

are inserted into a merkle tree. The root of the merkle tree is maintained on-chain. Tokens are stored at an address that is derived from the user's leaf in the merkle tree: stealthAddressBytes

=> bytes32ToAddress().

For transfering

a token, the contract requires a proof that a user can

i) generate a stealth address that is included in the merkle tree

ii) generate the merkle tree after updating the respective leaf

For minting

a token, the contract requires a proof that a user can

i) generate a stealth address and add it to an empty leaf in the merkle tree

ii) generate the merkle tree after updating the respective leaf

For burning

a token, the contract requires a proof that a user can

i) generate a stealth address and delete it from a leaf in the merkle tree

ii) generate the merkle tree after updating the respective leaf

NOTE:

The generation of the stealth address requires to have access to a private key. E.g. a user signs a message, the circuit parses the public key (…and address), hashes the address together with the token ID and a secret value and inserts the result into a leaf of the merkle tree. In the end the circuit compares the calculated and user-provided roots for verification.

For general information, have a look at Vitalik's short section on private POAPs inthis article on SoulBound tokens.

I think, this EIP is the exact implementation of what Vitalik described.

This is the current draft of the interface:

// SPDX-License-Identifier: CC0-1.0 pragma solidity ^0.8.6; ... interface ERC0001 */is ERC721, ERC165/* {

```
/// @notice Mints token to a stealth address `stA` if proof is valid. stA is derived from
///  stealthAddressBytes which is the MIMC Sponge hash `h` (220 rounds) of the user address `eoa`,
///  the token id `tid` and a user-generated secret `s`, such that stA <== address() <== h(eoa,tid,s).
/// @dev Requires a proof that verifies the following:
```

/// - prover can generate the StealthAddress (e.g. user signs msg => computePublicKey() => computeStealthAddress()). /// - prover can generate the merkle root from an empty leaf. /// - prover can generate the merkle root after updating the empty leaf. /// @param currentRoot A known (historic) root. /// @param newRoot Updated root. /// @param stealthAddressBytes Hash of user address, tokenId and secret. /// @param tokenId The Id of the token. /// @param proof The zk-SNARK. function _mint(bytes32 currentRoot, bytes32 newRoot, bytes32 stealthAddressBytes, uint256 tokenId, bytes proof) external;

```
/// @notice Burns token with specified Id from stealth address `stA` if proof is valid.
/// @dev Requires a proof that verifies the following:
///       - prover can generate the StealthAddress (e.g. user signs msg => computePublicKey() => computeStealthAddress() )
///       - prover can generate the merkle root from an non-empty leaf.
///       - prover can generate the merkle root after nullifieing the non-empty leaf.
/// @param currentRoot A known (historic) root.
/// @param newRoot Updated root.
/// @param stealthAddressBytes Hash of user address, tokenId and secret.
/// @param tokenId The Id of the token.
/// @param proof The zk-SNARK.
function _burn(bytes32 currentRoot, bytes32 newRoot, bytes32 stealthAddressBytes, uint256 tokenId, bytes proof) external;

/// @notice Transfers token with specified Id from current owner to the recipient's
/// stealth address, if proof is valid.
/// @dev Requires a proof that verifies the following:
///       - prover can generate the StealthAddress (e.g. user signs msg => computePublicKey() => computeStealthAddress() ).
///       - prover can generate the merkle root from an non-empty leaf.
///       - prover can generate the merkle root after updating the non-empty leaf.
/// @param currentRoot A known (historic) root.
/// @param newRoot Updated root.
/// @param stealthAddressBytes Hash of user address, tokenId and secret.
/// @param tokenId The Id of the token.
/// @param proof The zk-SNARK.
function _transfer(bytes32 currentRoot, bytes32 newRoot, bytes32 stealthAddressBytes, uint256 tokenId, bytes proof) external;

/// @notice Verifies zk-SNARKs
/// @dev Forwards the different proofs to the right `Verifier` contracts.
///  Different Verifiers are required for each action, because of the merkle-tree logic involved.
/// @param currentRoot A known (historic) root.
/// @param newRoot Updated root.
/// @param stealthAddressBytes Hash of user address, tokenId and secret.
/// @param tokenId The Id of the token.
/// @param proof The zk-SNARK.
/// @return Validity of the provided proof.
function _verifyProof(bytes32 currentRoot, bytes32 newRoot, bytes32 stealthAddressBytes, uint256 tokenId, bytes proof) external returns (bool);

}
```

This EIP is still in idea stage (no pull-request yet).

Looking for collaborators!