Taprootized Atomic Swaps

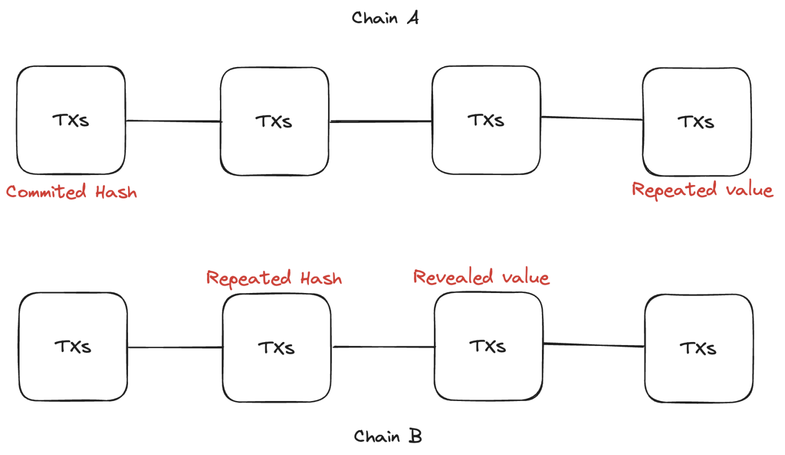
Cross-chain, Untraceable, Trustless

Distributed Lab, Jan 2024
Version 1.0

Abstract. Taprootized Atomic Swaps (TAS) is an extension for Atomic Swaps that presumes the untraceability of transactions related to a particular swap. Build top on Schnorr signatures, Taproot technology, and zero-knowledge proofs taprootized atomic swaps hide swap transactions under regular payments.

Intro

Atomic swap is an incredible approach to cross-chain exchanges without mediators. However, one of the disadvantages of its implementation in the classical form is the "digital trail" - any party can make a matching between transactions in the blockchains in which the exchange took place and find out both the participants in the exchange and the proportion in which assets were exchanged.

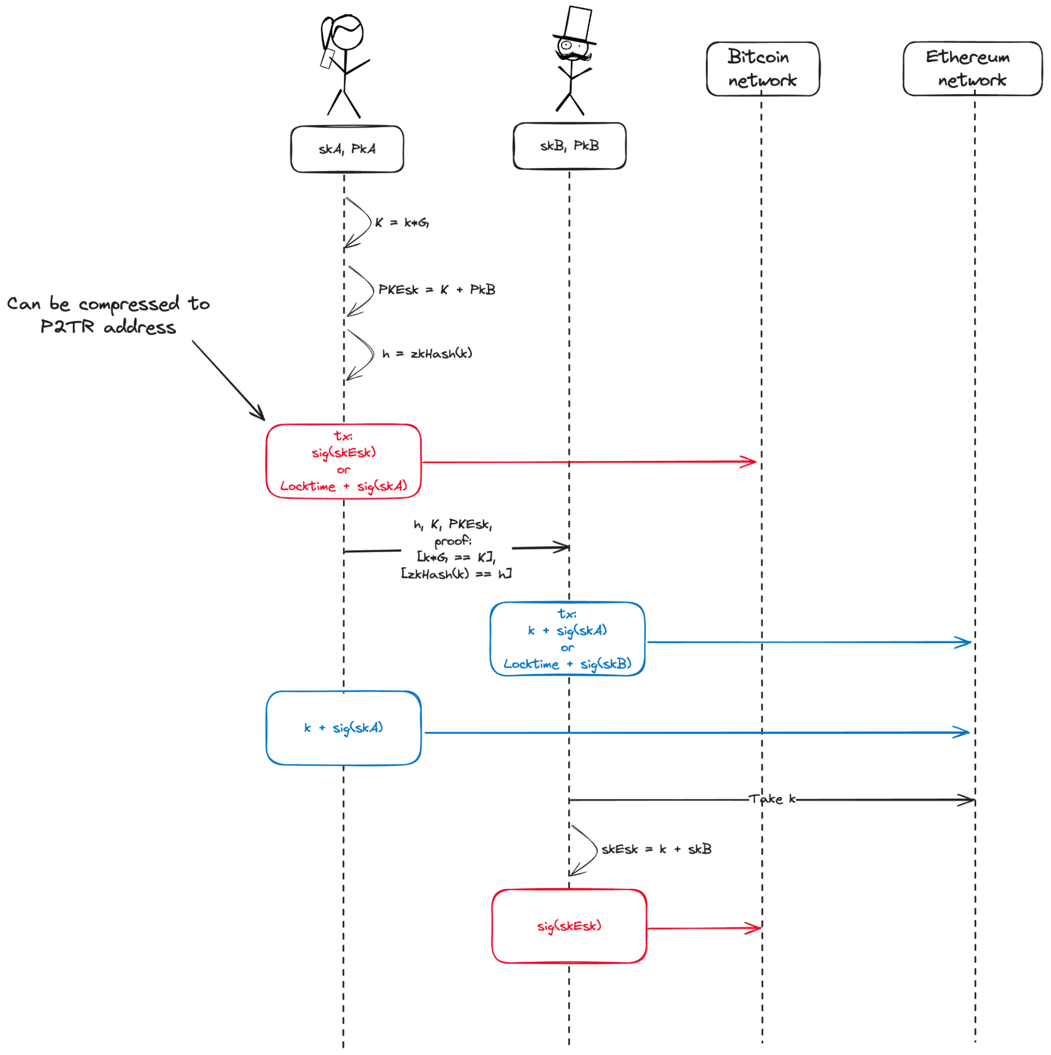


On the other hand, atomic swaps is a technology that initially assumed the involvement of only two parties and a "mathematical contract" between them directly. That is, an ideal exchange presupposes 2 conditions:

1) Only counterparties participate in the exchange (works by default)
2) Only counterparties know about the fact of the exchange (it would be nice to ensure)

This paper will provide a concept of taprootized atomic swaps that allow hiding the swap's very fact. To an external auditor, transactions to initiate and execute atomic swaps will be indistinguishable from regular Bitcoin payments. In the other accounting system involved in the transfer, more information is disclosed (the fact of exchange can be traced). Still, it is impossible to link this to the corresponding Bitcoin transactions (without additional context from the involved parties).

Protocol


Bitcoin network
Ethereum network
skA, PkA
skB, PkB
K = k*G
PKEsk = K + PkB
Can be compressed to P2TR address
h = zkHash(k)
tx: sig(skEsk) or Locktime + sig(skA)
h, K, PKEsk, proof: [k*G == K], [zkHash(k) == h]
tx: k + sig(skA) or Locktime + sig(skB)
k + sig(skA)
Take k
skEsk = k + skB
sig(skEsk)

The protocol includes the following steps:

1. Alice (sk_A, PK_A) and Bob (sk_B, PK_B) have their keypairs and know each other's public keys.
2. Alice generates a random $\boldsymbol{k}$ and calculates the public value $\boldsymbol{K = k * G}$
3. Alice calculates an escrow public key as $\boldsymbol{PK_{Esc} = K + PK_B}$
   a. The signature can be generated only with the knowledge of $\boldsymbol{k}$ and $\boldsymbol{sk_B}$
4. Alice calculates the $\boldsymbol{h}$ as a hash value of $\boldsymbol{k}$ (zk-friendly hash function is recommended to use)
5. Alice forms the funding transactions with the following conditions of how it can be spent:
   a. Signature of $\boldsymbol{sk_{Esc}}$: Bob, with knowledge of $\boldsymbol{k}$ and $\boldsymbol{sk_B}$ can spend the output
   b. Signature of $\boldsymbol{sk_A}$ + Locktime: Alice, with knowledge of $\boldsymbol{sk_A}$ can spend the output, but only after some point in time $\boldsymbol{t_1}$
6. Alice sends the transaction to the Bitcoin network
7. Alice generates the zero-knowledge ***proof*** that includes:
   a. The proof of knowledge of $\boldsymbol{k}$ that satisfies $\boldsymbol{k*G == K}$

    b. The proof of knowledge of $\boldsymbol{k}$ that satisfies ***zkHash(k)*** == $\boldsymbol{h}$
8. Alice provides the set of data to Bob:
    a. $\boldsymbol{h}$
    b. $\boldsymbol{K}$
    c. $\boldsymbol{PK_{Esc}}$
    d. ***proof***
9. Bob performs the following verifications:
    a. Verify that $\boldsymbol{PK_{Esc}} == \boldsymbol{K} + \boldsymbol{PK_B}$, it means that the valid PK of Bob was added to escrow PK
    b. Verifies that Alice knows $\boldsymbol{k}$ that satisfies $\boldsymbol{k*G} == \boldsymbol{K}$ and ***zkHash(k)*** == $\boldsymbol{h}$, it means that Bob can access the output $\boldsymbol{PK_{Esc}}$ if he receives $\boldsymbol{k}$
10. If verifications are passed, Bob forms the transaction that locks his funds on the following conditions:
    a. Publishing of $\boldsymbol{k}$ and the signature of $\boldsymbol{sk_A}$: only Alice can spend it if she reveals $\boldsymbol{k}$
    b. Signature of $\boldsymbol{sk_B}$ + Locktime: Bob, with knowledge of $\boldsymbol{sk_B}$, can spend the output, but only after some point in time $\boldsymbol{t_2}$
11. Bob sends the transaction to the Ethereum network (or other)
12. Alice sees the locking conditions defined by Bob and publishes the $\boldsymbol{k}$ together with the signature generated by her $\boldsymbol{sk_A}$. As a result - Alice spent funds locked by Bob.
    a. If Alice doesn't publish the relevant $\boldsymbol{k}$, Bob can return funds after locktime is reached
13. If Alice publishes a transaction with k, Bob can recognize it and extract the $\boldsymbol{k}$ value
14. Bob calculates the needed $\boldsymbol{sk_{Esc}}$ as $\boldsymbol{sk_{Esc}} = \boldsymbol{k} + \boldsymbol{sk_B}$
15. Bob sends the transaction with the signature generated by the $\boldsymbol{sk_{Esc}}$ and spends funds locked by Alice.

Implementation notes

1. As an approach for escrow public key forming, the usage of MuSig aggregation mechanism is preferable [1].
2. All conditions described in step 5 (Protocol section) can be put into a P2TR address. The formed address will not differ from the regular Bitcoin address (single or multisig) formed using the P2TR method [2].
3. As a zk-friendly hash function, we can use Poseidon [3].
4. For zk operations with EC points, we can use the 0xPARC library [4].

Links

[1] https://bitcoinops.org/en/topics/musig/

[2] https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

[3] https://github.com/iden3/circomlib/blob/master/circuits/poseidon.circom

[4] https://github.com/0xPARC/circom-ecdsa/blob/master/circuits/secp256k1.circom