Taprootized Atomic Swaps

Cross-chain, Untraceable, Trustless

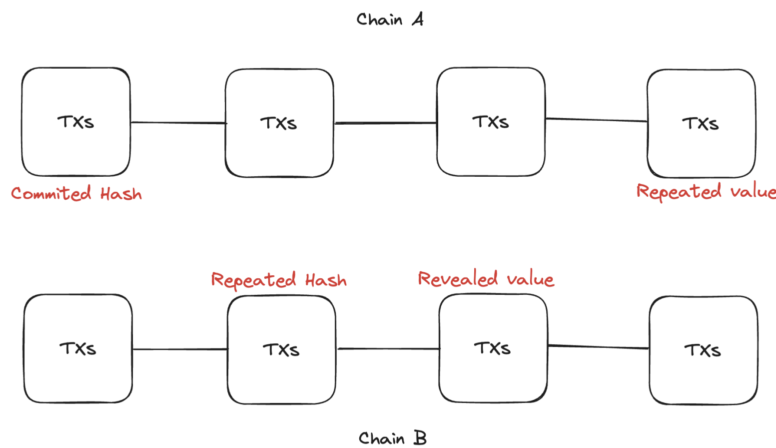
Distributed Lab, Jan 2024

Version 1.2

Abstract. Taprootized Atomic Swaps (TAS) is an extension for Atomic Swaps that enables the untraceability of transactions in a particular swap. Based on Schnorr signatures, Taproot technology, and zero-knowledge proofs, the taprootized atomic swaps hide swap transactions between 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 link the two transactions across blockchains where the swap occurred and find out both the participants of the swap and the proportion in which the assets were exchanged.

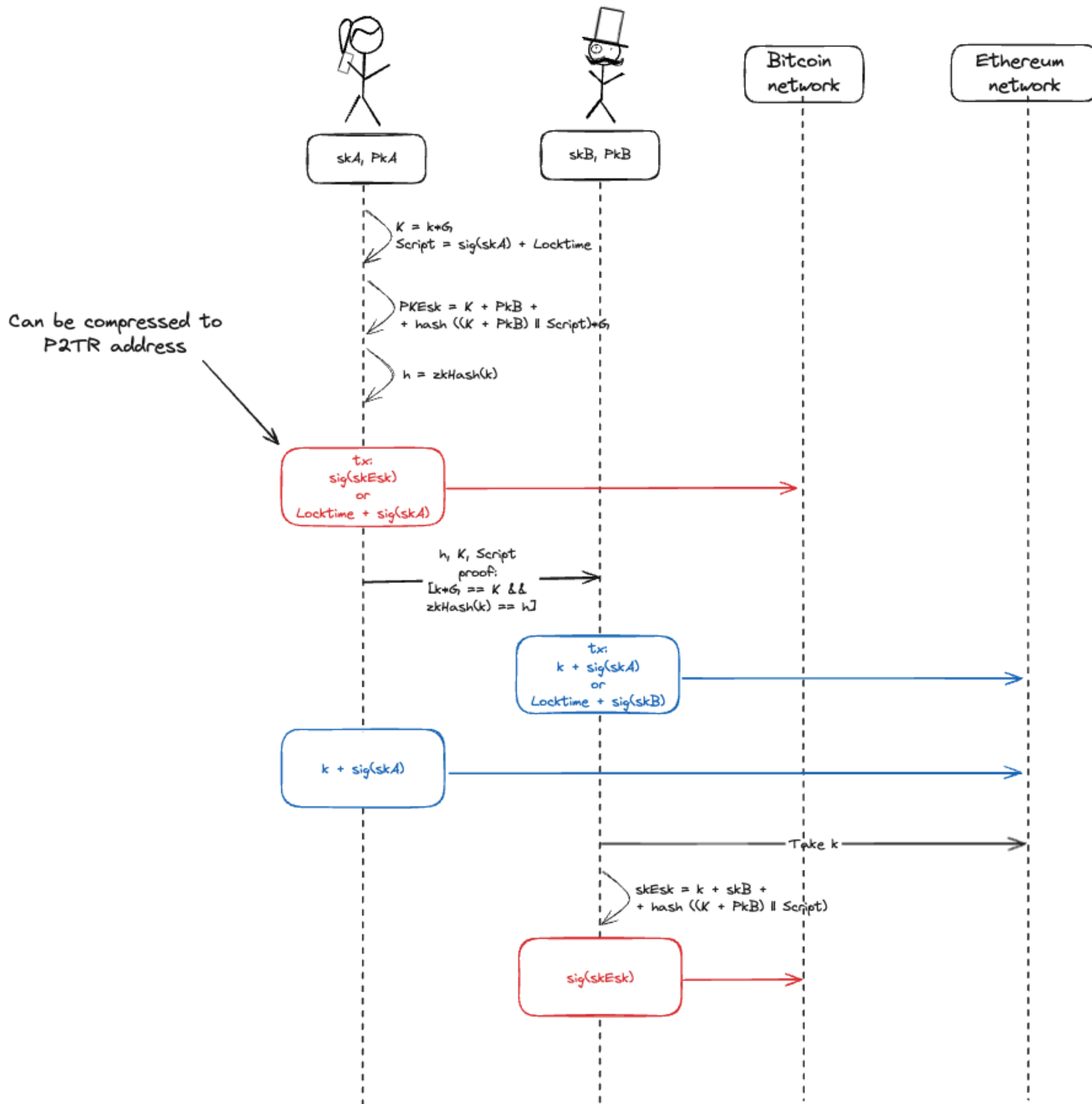


On the other hand, atomic swaps is a technology that initially assumed the involvement of only two parties, hence a “mathematical contract” between them directly. That is, an ideal exchange presupposes two conditions:

1) Only counterparties participate in the exchange (must-have)
2) Only counterparties can trace the exchange (nice-to-have)

This paper describes the design of the concept of taprootized atomic swaps, with the help of which it is now possible to conceal the very fact of the swap. To an external investigator/auditor, transactions that initiate and execute atomic swaps will appear indistinguishable from regular Bitcoin payments. In the other accounting system (i.e., blockchain) involved in the transfer, more information is disclosed (the fact of the swap can, in fact, be traced). However, it is impossible to link this to the corresponding Bitcoin transactions unless, obviously, the investigator has quite a specific insight from the involved parties (additional context can be provided by the time of the swap and approximate amount).

Protocol



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 forms the alternative spending path $\boldsymbol{Script = sig(sk_A) + Locktime}$ in the form of Bitcoin Script
4. Alice calculates an escrow public key as $\boldsymbol{PK_{Esc} = K + PK_B + hash((K + PK_B) \mid\mid Script) * G}$ (here, escrow is just a public key, formed using Taproot technology)
    a. The signature $\boldsymbol{sig(sk_{Esc})}$, verified by the $\boldsymbol{PK_{Esc}}$, can only be generated if Bob knows $\boldsymbol{k}$, $\boldsymbol{sk_B}$, and ***Script***
5. Alice calculates $\boldsymbol{h}$ as a hash value of $\boldsymbol{k}$ (zk-friendly hash function is recommended for use)
6. Alice forms the funding transactions and specifies the spending conditions:

    a. Signature of sk_{Esc}: Bob can spend the output only if he knows k, sk_B and ***Script***
    b. Signature of sk_A + Locktime: Alice can spend the output only if she knows sk_A and only after a certain point in time t_1 (this condition is the ***Script*** itself)
7. Alice sends the transaction to the Bitcoin network
8. Alice generates the zero-knowledge ***proof*** that includes (for the same k):
    a. The proof of knowledge of k that satisfies $k*G == K$
    b. The proof of knowledge of k that satisfies $zkHash(k) == h$
9. Alice provides the set of data to Bob:
    a. h
    b. K
    c. ***Script***
    d. ***proof***
10. Bob calculates PK_{Esc} as $K + PK_B + hash((K + PK_B) \| Script) * G$ and finds the transaction that locked BTC (to be precise, verifies it exists). Then Bob performs the following verifications:
    a. Verifies that Alice knows k that satisfies $k*G == K$ and $zkHash(k) == h$, meaning Bob can access the output PK_{Esc} if he receives k
    b. Verifies that the ***Script*** is correct and includes only the required alternative path.
11. If verifications are passed, Bob forms the transaction that locks his funds on the following conditions:
    a. Publishing of k and the signature of sk_A: only Alice can spend it if she reveals k (hash preimage)
    b. Signature of sk_B + Locktime: Bob, if he knows sk_B, can spend the output, but only after a certain point in time t_2
12. Bob sends the transaction to the Ethereum network (or any other that supports ***zkHash()***)
13. Alice sees the locking conditions defined by Bob and publishes the k together with the signature generated by her sk_A. As a result, Alice spends funds locked by Bob.
    a. If Alice doesn't publish the relevant k, Bob can return funds after t_2 is reached
14. If Alice publishes a transaction with k, Bob can recognize it and extract the k value
15. Bob calculates the needed sk_{Esc} as $sk_{Esc} = k + sk_B + hash((K + PK_B) \| Script)$
16. Bob sends the transaction with the signature generated by the 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