PolkaBTC Documentation

Release 0.0.1

Interlay

Nov 20, 2019

CONTENTS:

CHAPTER

ONE

INTRODUCTION

CHAPTER

TWO

OVERVIEW OF CHAIN RELAYS

2.1 What are Chain Relays?

A chain relay is a program or smart contract deployed on a blockchain, which provides functionality for verifying the state of another blockchain.

2.2 Verification vs. Validation

2.3 Necessary Assumptions

2.4 Recommended Background Reading

- **Enabling Blockchain Innovations with Pegged Sidechains**. *Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra A., Timon J., & Wuille, P*. (2019)
- **XCLAIM: Trustless, Interoperable, Cryptocurrency-backed Assets**. *IEEE Security and Privacy (S&P).* Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., & Knottenbelt, W. (2019).
- **SoK: Communication Across Distributed Ledgers**. *Cryptology ePrint Archiv, Report 2019/1128*. Zamyatin A, Al-Bassam M, Zindros D, Kokoris-Kogias E, Moreno-Sanchez P, Kiayias A, Knottenbelt WJ. (2019)
- **Proof-of-Work Sidechains**. *Workshop on Trusted Smart Contracts, Financial Cryptography* Kiayias, A., & Zindros, D. (2018)

CHAPTER

THREE

BTC-PARACHAIN DESIGN AND MODEL

3.1 System Model

- Parachain Validators
- Collators (?)

(Make sure we use same nomenclature as Polkadot)

3.2 Network Model

- Communication model assumptions.
- Need to assume a sync. communication model
- Define upper bounds on timeouts

3.3 Threat Model

- What can an attacked do?

(maybe move this to a separate chapter, where we discuss security?)

3.4 Architecture: Storage vs. Verification

- Storage component
- Verification component

3.5 Cryptographic Primitives

Bitcoin's Cryptographic Primitives

- ECDSA secp256k1
- SHA-256 hash function
- RIPEMID-160 hash function

CHAPTER

FOUR

STORAGE COMPONENT: DATA MODEL FOR BITCOIN

4.1 Block Headers

4.2 Transactions

4.2.1 Inputs

4.2.2 Outputs

4.2.3 Merkle Tree Paths

4.3 Data Format Considerations

- Endianness
- Specific Bitcoin data types and structs (e.g. Merkle Block)

CHAPTER

FIVE

VERIFICATION COMPONENT: PROCESS AND METHODS

5.1 Step-by-Step Verification Process

5.2 Verification Functionality

5.2.1 Correct Difficulty Target

5.2.2 Sufficient Proof-of-Work

5.2.3 Valid Chain

5.2.4 Main Chain

5.2.5 Transaction Inclusion

5.3 Transaction Parsing

CHAPTER

SIX

DATA TYPES

CHAPTER

SEVEN

FUNCTIONS

7.1 setInitialParent

7.2 storeBlockHeader

7.3 verifyTransaction

The `verifyTransaction` function is one of the core components of the chain relay: this function returns whether a given transaction is valid by considering a number of parameters. The core idea is that a user submits a transaction hash including the parameters to proof to another party that the transaction is included in the Bitcoin blockchain. Since the verification is based on the data in the chain relay, other parties can rely on the trustworthiness of such a proof.

7.3.1 Process

Generally, a user has to follow four steps to successfully verify a transaction:

1. The user ensures that the Bitcoin block header, in which his transaction is included, is stored in the BTCRelay (see *storeBlockHeader*).
2. The user ensures that the block has the minimum number of required confirmations (typically `6`).
3. The user prepares the necessary input parameters from the Bitcoin blockchain to call the `verifyTransaction` function. The user can receive these parameters from the bitcoin-rpc client.
    a. The *transaction hash* is the transaction that should be verified. The user should note the transaction hash when sending a Bitcoin transaction he wants to verify.
    b. The *block height* refers to the block in which the transaction is included. The user receives the block height from the getrawtransaction `bitcoin-rpc` method by querying for his transaction hash and receiving the `blockindex`.
    c. The *transaction index* specifies the index of the transaction in the block. The user receives the index from the `bitcoin-rpc` method getblock by geting the index from the `tx` array storing the all transaction hashes in that block.
    d. The *Merkle proof* encodes how to calculate the Merkle root from the transaction hash. The user receives the proof from the `bitcoin-rpc` method gettxoutproof.
4. The user submits the above parameters to the `verifyTransaction` function and receives one of two possible results.
    a. `True`: the transaction is successfully verified.
    b. `False`: the transaction cannot be verified given the input parameters provided by the user.

7.3.2 Conditions

A transaction is successfully verified if the following conditions are met.

1. The user submits a valid *transaction hash*. The transaction hash is 32 bytes long.
2. The submitted *block height* is stored in BTCRelay.
3. The block in which the transaction is included has enough confirmations (default `6`).
4. The user submitted a valid *Merkle proof*. The Merkle proof needs to contain the *transaction hash* in its first 32 bytes. Further, the last hash in the Merkle proof must be the block header hash in which the transaction is included.
5. The *Merkle proof* parses correctly. The *transaction hash* is combined with each hash in the Merkle proof until the resulting hash must equal the Merkle root. Details on this are included in the Bitcoin developer reference.

7.3.3 Use Cases

Issue of Bitcoin-backed Assets: Users can create Bitcoin-backed tokens on Polkadot by proving to the Polkadot blockchain that they have sent a number of Satoshis to a vault's Bitcoin address. To realize this, a user acts as a so-called CbA Requester. First the CbA-Requester transfers the Satoshis to the Bitcoin address of a Vault on the Bitcoin blockchain. The CbA-Requester notes the transaction hash of this transaction. Next, the CbA-Requester proves to the Polka-BTC bridge that the vault has received his Satoshis. He achieves this by ensuring that the block header of his transaction is included in the BTCRelay and has enough confirmations. He then extracts the input parameters as described in step 3 of the *Process* above. With these input parameters he calls the `verifyTransaction` to receive a successful transaction inclusion proof.

7.3.4 Implementation

Function Signature

`verifyTransaction(txId, txBlockHeight, txIndex, merkleProof)`

Parameters

- `txId`: the hash of the transaction.
- `txBlockHeight`: block height at which transacton is supposedly included.
- `txIndex`: index of transaction in the block's tx Merkle tree.
- `merkleProof`: Merkle tree path (concatenated LE sha256 hashes).

Returns

- `True`: if txId is at the claimed position in the block at the given txBlockHeight.
- `False`: otherwise.

Events

- `VerifyTransaction(txId, blockHeight, result)`: issue an event for a given txId and a blockHeight and return the result of the verification (either `True` or `False`).

Errors

- `ERR_INVALID_TXID = "Invalid transaction identifier"`: raise an exception when the transaction id (`txId`) is malformed.
- `ERR_CONFIRMATIONS = "Transaction has less confirmations than requested"`: raise an exception when the number of confirmations is less than required.

- `ERR_MERKLE_PROOF = "Invalid Merkle Proof structure"`: raise an exception when the Merkle proof is malformed.

7.4 Helper Methods

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search