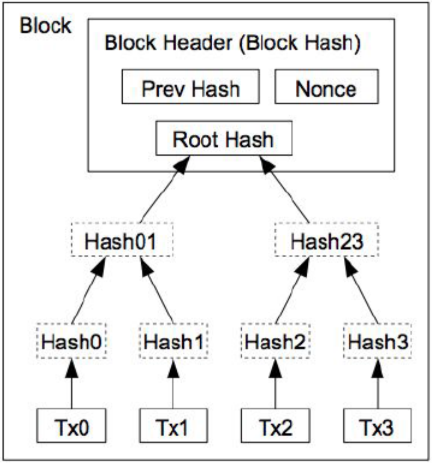
IT486 v3.0: Blockchains and Cryptocurrencies

Simplified Payment Verification (SPV)

Recap: Structure of a block



Transactions Hashed in a Merkle Tree

Types of nodes

- Miner nodes
  - compute and propose new blocks to the blockchain
  - run on powerful or specialized hardware

Types of nodes

- Miner nodes
  - compute and propose new blocks to the blockchain
  - run on powerful or specialized hardware
- Full nodes
  - validate blocks mined by miners and verify transactions
  - store the full copy of the blockchain with all the transactions (about 300 GB, September 2020)

Thin clients

- Full nodes have large communication, computation and storage overheads
- Thin clients are more viable for less powerful/mobile devices

Thin clients

- Full nodes have large communication, computation and storage overheads
- Thin clients are more viable for less powerful/mobile devices
    - also called Simplified Payment Verification (SPV) nodes

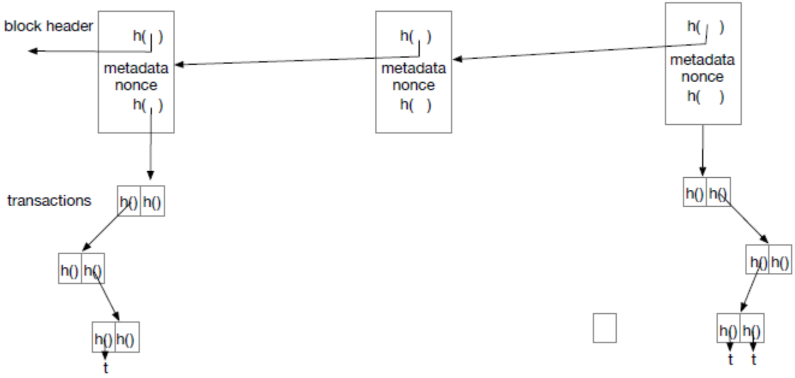
Thin clients

- Full nodes have large communication, computation and storage overheads
- Thin clients are more viable for less powerful/mobile devices
  - also called Simplified Payment Verification (SPV) nodes
  - don't store a full copy of blockchain, only store the block headers

Thin clients

- Full nodes have large communication, computation and storage overheads
- Thin clients are more viable for less powerful/mobile devices
    - also called Simplified Payment Verification (SPV) nodes
    - don't store a full copy of blockchain, only store the block headers
    - check only transactions they care about by querying a full node

Data maintained by SPV nodes



SPV: Verifying transactions

- The SPV node asks a full node to search for the txn of interest
- The full node responds with
  - the block header hash of the block the transaction is included in
  - the Merkle path of the transaction

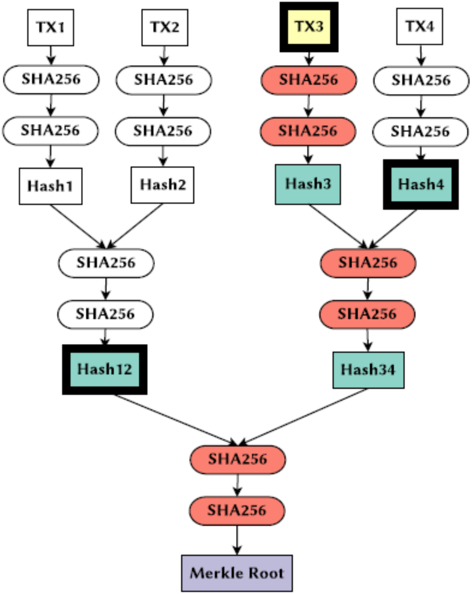
SPV: Verifying transactions

- The SPV node asks a full node to search for the txn of interest
- The full node responds with
  - the block header hash of the block the transaction is included in
  - the Merkle path of the transaction
- The SPV node can identify the block using its block header hash; it already has that info

SPV: Verifying transactions

- The SPV node asks a full node to search for the txn of interest
- The full node responds with
  - the block header hash of the block the transaction is included in
  - the Merkle path of the transaction
- The SPV node can identify the block using its block header hash; it already has that info
- The Merkle root is also part of the block header, so the SPV node already has that too

SPV: Verifying transactions



SPV: Verifying transactions

- The SPV node needs to rehash the txn and verify the Merkle root that is stored in the block header
- Using the Merkle path, it only downloads the relevant hashes from the full node

SPV: Verifying transactions

- The SPV node needs to rehash the txn and verify the Merkle root that is stored in the block header
- Using the Merkle path, it only downloads the relevant hashes from the full node
- The SPV node does not have to know anything about the other transactions in the block

Security issues

- A SPV node can verify that a txn is actually included in a block

Security issues

- A SPV node can verify that a txn is actually included in a block
- A SPV node cannot be convinced that txn exists if the txn doesn't really exist

Security issues

- A SPV node can verify that a txn is actually included in a block
- A SPV node cannot be convinced that txn exists if the txn doesn't really exist
- A txn can be "hidden" from a SPV node
  - DoS attacks
  - Double spend attacks

DoS attack on SPV node

- F: full node, C: SPV node
- Attack strategy
  - F does not report all transactions making payments to addresses controlled by C

DoS attack on SPV node

- F: full node, C: SPV node
- Attack strategy
  - F does not report all transactions making payments to addresses controlled by C
  - C thinks it has less money than it has

Double spend attack on SPV node

- F: full node, C: SPV node
- Attack strategy
  - F can partition the network, thereby control which messages go where

Double spend attack on SPV node

- F: full node, C: SPV node
- Attack strategy
  - F can partition the network, thereby control which messages go where
  - F creates a txn transferring the money to C, presents it to C, but not to the network

Double spend attack on SPV node

- F: full node, C: SPV node
- Attack strategy
  - F can partition the network, thereby control which messages go where
  - F creates a txn transferring the money to C, presents it to C, but not to the network
  - F creates another txn transferring the money to his second account, but hides it from C

Defence against attacks on SPV nodes

- The main defence is to not rely on a single full node
- Call k randomly selected full nodes, and cross-check the answers

Defence against attacks on SPV nodes

- The main defence is to not rely on a single full node
- Call k randomly selected full nodes, and cross-check the answers
- this decreases the probability of an attack to probability a randomly selected node is malicious
    - which very quickly becomes a very small number as k increases

Privacy issue with SPV nodes

- SPV nodes that request only transactions for addresses they own reveal those addresses to the full nodes they connect to

Privacy issue with SPV nodes

- SPV nodes that request only transactions for addresses they own reveal those addresses to the full nodes they connect to
- Bloom filters have been to added to SPV clients to mitigate this risk

Bloom Filters

- Bloom filters are data structures that allow you to:
  - insert an element into a set
  - query whether an element is in a set

Bloom Filters

- Bloom filters are data structures that allow you to:
  - insert an element into a set
  - query whether an element is in a set
- When the answer to the query is “yes”, it can be wrong (false positive!)

Bloom Filters

- Bloom filters are data structures that allow you to:
  - insert an element into a set
  - query whether an element is in a set
- When the answer to the query is "yes", it can be wrong (false positive!)
- "no" answers are always correct (zero false negatives)

How SPV nodes use bloom filters

- The SPV node
  - creates an empty bloom filter
  - makes a list of everything it is interested in
    - addresses, keys, hashes
  - adds all of these to the bloom filter
- The SPV node sends the bloom filter to the full node

How SPV nodes use bloom filters

- The SPV node
  - creates an empty bloom filter
  - makes a list of everything it is interested in
    - addresses, keys, hashes
  - adds all of these to the bloom filter
- The SPV node sends the bloom filter to the full node
- The full node checks the bloom filter against the blockchain
  - only txns that match the bloom filter are sent back to the SPV node

How SPV nodes use bloom filters

- For each matching txn, the full node sends back
  - block headers, Merkle paths, txn messages
- The SPV node
  - discards all irrelevant info
  - uses the matching txns to update its UTXO set and wallet balance

Bloom Filter Example

The data structure of the Bloom Filter is a bit array:

1	0	1	1	0	1	1	1

Bloom Filter Example

Start with an empty bit array (all zeros), and k hash functions.

$k1 = (13 - (x \,\%\, 13)) \,\%\, 7, \;\; k2 = (3 + 5x) \,\%\, 7,$ etc.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Bloom Filter Example

Values to be inserted then get hashed by all k hashes, and the bit in the position is set to 1 in each case.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Bloom Filter Example

Insert 129: x=129

$k1 = (13 - (x \,\%\, 13)) \,\%\, 7, \;\; k2 = (3 + 5x) \,\%\, 7,$ etc.

0	1	2	3	4	5	6	7
0	**1**	0	0	**1**	0	0	0

$k1$ == 1, so we change bit 1 to a 1
$k2$ == 4, so we change bit 4 to a 1

Bloom Filter Example

Insert 479: x=479

$k1 = (13 - (x \,\%\, 13)) \,\%\, 7, \;\; k2 = (3 + 5x) \,\%\, 7,$ etc.

0	1	2	3	4	5	6	7
0	**1**	**1**	0	**1**	0	0	0

$k1$ == 2, so we change bit 2 to a 1
$k2$ == 4, so we would change bit 4 to a 1,
but is already a 1

Bloom Filter Example

To check if 129 is in the table, just hash again and check the bits.
$k1 = 1, \;\; k2 = 4:$ probably in the table!

0	1	2	3	4	5	6	7
0	**1**	**1**	0	**1**	0	0	0

$k1 = (13 - (x \,\%\, 13)) \,\%\, 7, \;\; k2 = (3 + 5x) \,\%\, 7,$ etc.

Bloom Filter Example

To check if 123 is in the table, hash and check the bits.

$k1 = 0, \;\; k2 = 2:$ cannot be in table because 0 bit is still 0

0	1	2	3	4	5	6	7
0	**1**	**1**	0	**1**	0	0	0

$k1 = (13 - (x \,\%\, 13)) \,\%\, 7, \;\; k2 = (3 + 5x) \,\%\, 7,$ etc.

Bloom Filter Example

To check if 402 is in the table, hash and check the bits.

$k1 = 1, \; k2 = 4:$ Probably in the table (but isn't, a false positive!)

0	1	2	3	4	5	6	7
0	**1**	**1**	0	**1**	0	0	0

$k1 = (13 - (x \,\%\, 13)) \,\%\, 7, \;\; k2 = (3 + 5x) \,\%\, 7,$ etc.