

chaincode

# Security models

Chaincode Residency, June 17th 2019


# Security assumptions

---

- Security proofs are often made in terms of existing schemes:
  - if you can break new scheme B, then you'd be able to break old scheme A, which we assume is hard.
- eg: under the random oracle model, if you could break taproot, then you could break DLP in general
- Why do we think that DLP is hard?
- Because we haven't broken it yet!



# Secure compared to what?



Can we lower the cost of  
interacting with the Bitcoin  
network while maintaining an  
acceptable security level?

The background features abstract geometric shapes. On the left, a dark gray triangle points downwards, overlapping a yellow triangle that also points downwards. To the right, a large yellow trapezoidal shape is positioned, with the word 'Agenda' centered within it.

# Agenda

- Full and pruned nodes
- Light clients
- Checkpoints, assumevalid and assumeutxo
- Alternative UTXO set proposals
- Further reading



Full and pruned  
nodes

# Full node

---

- Downloads all headers and blocks
- Validates all blocks and transactions in the most-work blockchain
- May validate and relay unconfirmed transactions



# Pruned node

---

- Downloads and validates all blocks and headers
- Discards all block files over a certain storage limit

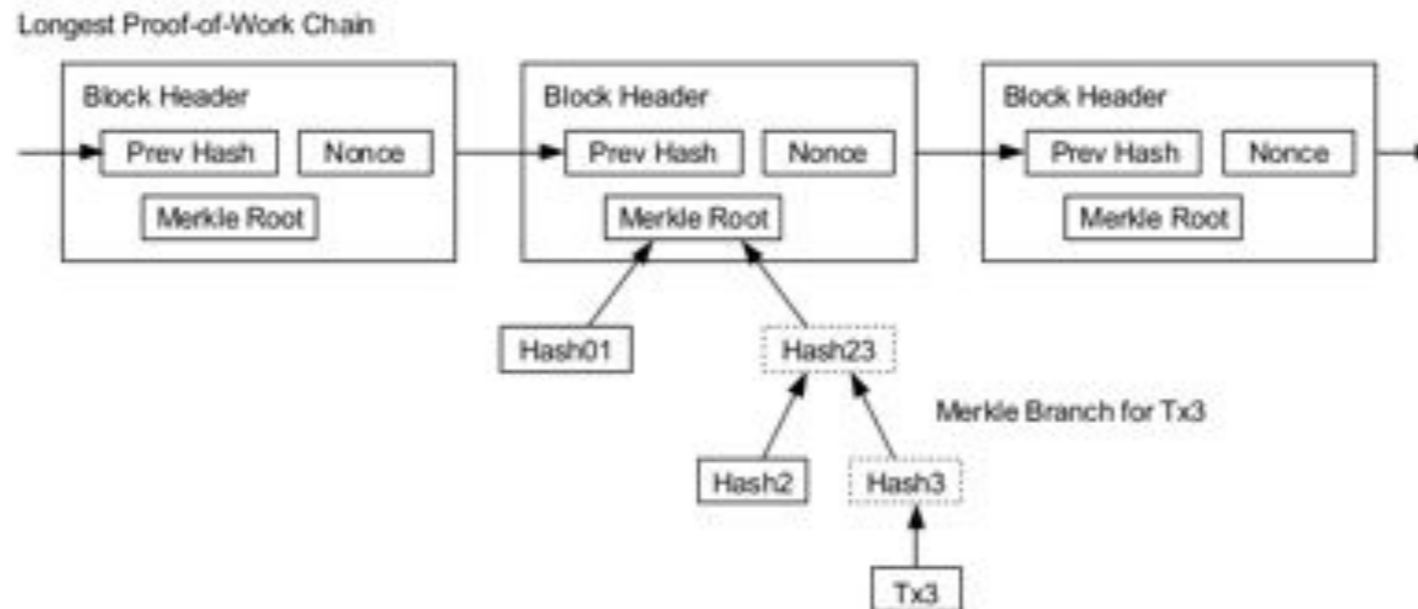
The background features abstract geometric shapes. On the left, a dark gray diagonal band runs from the top-left towards the bottom-right. To its right, a large yellow shape is composed of a horizontal bar at the top and a trapezoidal section below it, extending from the left edge towards the center. The remaining space is white.

Light clients

# SPV Nodes

## 8. Simplified Payment Verification

It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.



As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.

*It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.*

*As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.*

# SPV nodes - what they can do

---

- Can verify that a transaction has been confirmed by a certain amount of work
- Know what the most accumulated work chain is

# SPV nodes - what they can't do

---

- Determine whether the block, or the transaction chain is valid
- Enforce *your* set of consensus rules
- ... and so ensure that your preferred monetary policy/inflation schedule is being followed
- Give you the same level of privacy
- Detect false negatives
- Fee estimation

# But what about Satoshi's Vision?



# Bloom filters

---


- Defined in BIP 37
- Implemented in Bitcoin Core in August 2012
- Allows a light client user to request their transactions without revealing everything about their addresses
- Uses probabilistic filters so there are false positives to conceal the user's addresses/keys
- Not very effective at preserving privacy
- Places load on the server. Can be used as to DOS servers providing the filters
- Doesn't work with segwit
- Generally not advised. Will probably be disabled by default in the next Bitcoin Core release



# Compact Block Filters

---

- Defined in BIPs 157/158
- Flips the BIP 37 protocol. Instead of a server creating a unique filter for every client, the server creates one filter that is served to all clients
- Uses Golomb-Rice coding for compression
- Fully implemented in btcd
- Compact Block Filters and index are implemented for Bitcoin Core. P2P implementation is WIP



Checkpoints,  
assumevalid and  
assumeutxo

# Origin of checkpoints

---

- Satoshi added checkpoints in 0.3.2 to prevent 51% attack from reorging the blockchain too far (200 blocks prior to release)
- Three checkpoints added:
  - 11111
  - 33333
  - 68555
- What it actually did: you can't connect a block at height 68555 (or 33333, or 11111) unless it matches the prescribed hash.
- What it didn't do: provide anti-DoS protections, or performance optimizations when syncing the prescribed chain.

# Changes to checkpoints

---

- At some point, checkpoints started to be used to as an optimization during IBD. For blocks below the last checkpoint height, we would skip validation.
  - That was later changed to only skip script validation for ancestors of the checkpointed block
- Then checkpoints also started to be used as an anti-DoS measure; don't process block headers that were timestamped older than the last checkpointed time.



# Evolution of thinking

---

- Stopped adding checkpoints at 295,000 (July 2014)
- Generally, there's concern philosophically about developer centralization/control if we keep pushing out new checkpoints.
- Separated checkpoints from signature skipping during IBD in 0.14, replaced with `assumevalid`.
- Started to use a new metric, `nMinimumChainWork`, for more anti-DoS protections, in places where checkpoints used to be used.
- Checkpoints still lock in the chain as a defense against low-difficulty headers being used to overwhelm a node.

# Moving forward

---

What do we need to do in order to finally get rid of checkpoints?

- Need a defense against a low difficulty headers attack (memory blowup).
- One approach: soft fork in a min-difficulty that is higher than current.
- Another approach: p2p only changes. Can be done with new p2p messages.

# assumevalid

---

- Specifies a block whose ancestors we assume all have valid scriptSigs
- Unlike checkpoints this has no influence on consensus (unless you set it to a block with an invalid history)
- Old releases with defaults will sync slower, but the value is configurable
- Is this trusting the developers?



*With assumevalid, you assume that the people who peer review your software are capable of running a full node that verifies every block up to and including the assumedvalid block. This is no more trust than assuming that the people who peer review your software know the programming language it's written in and understand the consensus rules; indeed, it's arguably less trust because nobody completely understands a complex language like C++ and nobody probably understands every possible nuance of the consensus rules---yet almost anyone technical can start a node with -noassumevalid, wait a few hours, and check that bitcoin-cli getblock \$assume\_valid\_hash returns has a "confirmations" field that's not -1.*

# assumeutxo

---

- The UTXO set at a certain height is snapshotted, and a hash commitment to the serialized set (+ block header hash) is made.
- The client takes the *assumeutxo commitment*, downloads the serialized UTXO set, syncs the headers chain to the committed block header, and then continues initial sync from that point to the tip.
- Once initial sync is complete, the node downloads the blocks from genesis to the *assumeutxo* block and verifies that the UTXO sets match.

# assumeutxo (cont)

---

- assumeutxo values could come from several places:
  - Provided as config option or RPC input
  - Hardcoded into the source code and reviewed as part of the release cycle
  - Committed to in coinbase transaction (?)
- Full UTXO set is then downloaded from one or more sources:
  - Initially out-of-band
  - In future, over the P2P network



# Alternative UTXO set proposals

# Alternative UTXO proposals

---

- Alan Reiner's 'trust-free lite nodes' (2012)
- Peter Todd's TXO MMR commitments (2016)
- Bram Cohen's TXO bitfield (2017)
- Pieter Wuille's Rolling UTXO set hashes (2017)

# trust-free lite nodes (Reiner)

---

- Proposed by Alan Reiner in a bitcointalk post
- There is a second chain that commits to the UTXO set, which is merge-mined with the main chain.
- The commitment is some kind of tree, perhaps a binary search tree, or a patricia tree, or a de-la-briandais tree, or a red-black tree.
- Lots of discussion about what this structure should be. If the UTXO is to be committed to, it should be fast to update (or committed to in a later block)
- Lite client stores the root of this structure and asks for proofs-of-inclusion / proofs-of-exclusion for their outputs.

# trust-free lite nodes (cont)

---

- You can just download a utxo set from somewhere, and check using the latest block that it's valid (or, say, 1000 blocks back, depending on how certain you want to be).
- This discussion evolved towards the root being a commitment instead of an alt-chain
- Pieter Wuille had implemented ultraprune at roughly the same time. He suggested that Ultraprune could be thought as a step towards these ideas

# TXO MMR commitments (Todd)

---

- A new hash root is committed to in the block. The structure is a Merkle Mountain Range:
  - deterministic
  - indexable
  - insertion ordered
- New items can be cheaply appended to the tree
- Once items are added, they are never removed, just updated in place.
- The state of a specific item in the MMR, as well as the validity of changes to items in the MMR, can be proven with  $\log_2(n)$  sized proofs consisting of a merkle path to the tip of the tree.



# TXO MMR commitments (Todd)

---

- At an extreme, with TXO commitments we could even have no UTXO set at all.
- A more realistic implementation is to have a UTXO cache for recent transactions.
- The spender provides the proof that the TXO exists and is unspent.
- Proofs can be generated and added to transactions after the fact
- The commitment is delayed - ie in block  $i$  the miner commits to the TXO set in block  $i-n$  where  $n$  is some system constant.

# TXO MMR commitments (Todd)

---

- A later post suggested that the TXO commitment doesn't need to be committed at all
- A full node that doesn't have enough local storage to maintain the full UTXO set can instead keep track of a TXO commitment, and prune older UTXO's from it
- In the event those UTXO's are spent, transactions and blocks spending them can provide proofs to temporarily fill-in the node's local TXO set database

# TXO bitfield (Cohen)

---

- Wallets maintain a proof of position for each output they want to spend (a simple merkle proof off some block, but with an output number rather than just a tx number), which is immutable.
- Nodes maintain a TXO bitset, which indicates what outputs are spent: even implemented as a naive bit array, this is  $1/256$  the size of the full TXO set
- Over time, the bitfield becomes sparse, and can be compressed quite compactly.
- This is not a block consensus change, it's merely a peer-to-peer upgrade.

# Rolling UTXO set hash (Wuille)

---

- Concatenating all the UTXOs in a canonical order and then hashing is simple, but expensive to update.
- Efficiency requires that we be able to hash the data in any arbitrary order, and remove as well as add elements.
- Proposal initially not for committing data. Use cases are:
  - Replacement for Bitcoin Core's `gettxoutsetinfo` RPC's hash computation.
  - Assisting in implementation of schemes like `assumeutxo`
  - Database consistency checking

# Rolling UTXO set hash proposals

---

- A rolling, incremental hash of the set of objects.
- 'Hashing onto a curve point' on the libsecp256k1 curve. Hash each item in the set onto the curve, and then sum them under elliptic curve addition
- Lthash - homomorphic hashing

# Rolling, incremental hash

---

- Bellare-Micciancio paper “A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost” suggests ways of incremental hashing:
  - XHASH - hash individual objects and XOR - is trivially insecure under Wagner’s attack
  - AdHASH - hash individual objects and add modulo some large prime - is also insecure under Wagner’s attack.
  - MuHASH - hash individual objects and multiply modulo some large prime - is secure under the DL assumption.
- If we use MuHash, we remove an item by finding the inverse of its hash under multiplication modulo the prime.

# Hashing onto a curve

---

- Elliptic Curve Multiset Hash is efficient, but uses a strange binary elliptic curve
- One other approach is just reading potential X coordinates from a PRNG until one is found that has a corresponding Y coordinate according to the curve equation.  
On average, 2 iterations are needed.
- Then add the points under EC point addition. To remove a point from the set, add its inverse (same point with Y co-ordinate inverted)

# LtHash

---

- Developed by facebook security team
- All data elements are hashed to a 2KB digest
- Two digests can be 'added' by breaking up each output into 16-bit chunks and performing component-wise vector addition modulo  $2^{16}$ .
- Properties:
  - Set homomorphic
  - Collision resistant



The background features abstract geometric shapes. On the left, a dark grey triangle points downwards, overlapping a yellow triangle that points upwards. To the right, a large yellow trapezoidal shape is positioned, containing the text.

Further reading

# Further reading

---

- **UHS** - Full-node security without maintaining a full UTXO set
- **utreexo** - A dynamic accumulator for Bitcoin state
- **Accumulators** - A scalable drop-in for Merkle Trees
- **flyclient** - Super-Light Clients for Cryptocurrencies

**Questions?  
Comments?**