Security Models

Intro - Security assumptions

In cryptography, security assumptions for new cryptographic schemes are often made in terms of existing schemes eg if you can break new scheme B, then you'd be able to break old scheme A, which we assume is hard.

Example: if you can forge a Schnorr signature, then you can break the DLP over our elliptic curve.

But why do we consider DLP to be hard? Because we haven't broken it yet.

The point is that when speaking about security models, you always have to ask "compared to what?"

This talk is about different security models so I think that's a good framework to use. We'll start from a node that validates everything, and compare other constructions to that.

Full node - the gold standard

- Downloads all headers and blocks
- Validates all blocks and transactions
- Is `-blocksonly` a full node?

Pruned node

- As above, but discards block files over a certain capacity
- Is this a full node?
- What are the additional security assumptions?

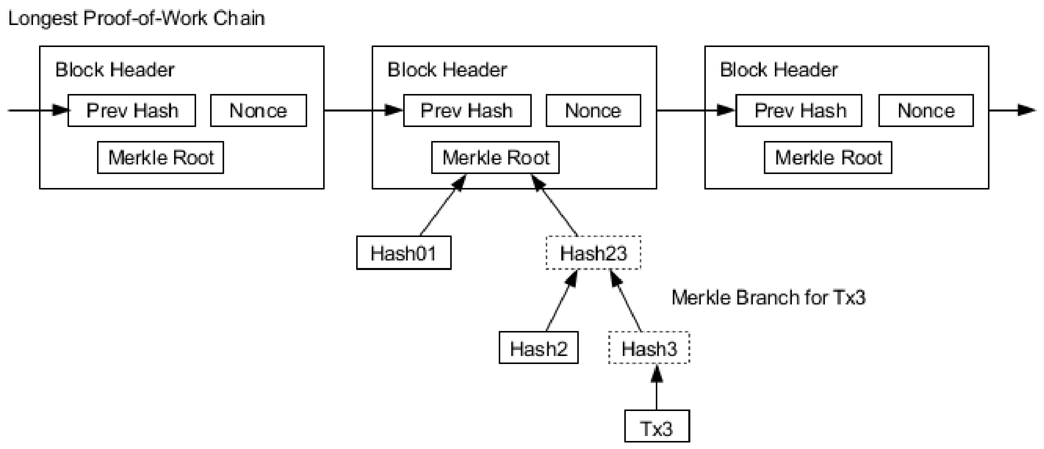
Light Nodes

Simplified Payment Verification

- The term 'SPV' was introduced by Satoshi in the Bitcoin Whitepaper:

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 therefore:

- Can verify that a transaction has been confirmed by a certain amount of work
- Know what the most accumulated work chain is (assuming they have one honest peer)

What they can't do:

- Determine whether the block, or the transaction chain is valid.
    - Satoshi suggested was '*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.*'
        - Why is this infeasible?
        - (you can't protect against spam alerts that induce you to download the entire blockchain).
    - Another suggestion is 'fraud proofs', which would be a compact cryptographic proof that a block is invalid. Again, in practice it seems that this is difficult to do in the general case.
    - Luke-jr proposed a method for doing this in the specific case of a too-large block: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-March/013756.html , but the proposal seems to have been removed now.
- Enforce *your* set of consensus rules. Most of the time 'your set of consensus rules' == 'valid' above, but there are times where there isn't universal consensus for consensus. In that case, you're not able to enforce which side of the fork you're on.
    - Segwit2x proponents were relying on this behaviour to make sure that SPV nodes followed their chain (https://medium.com/@jgarzik/why-segwit2x-is-the-best-path-for-bitcoin-d9a4103f2fda#wallet-compatibility)
    - BCH nodes could not use this behaviour because they have a different sighash tx digest algorithm (BIP 143)
- (part of the above). Ensure that your preferred monetary policy/inflation schedule is being followed. This is particularly relevant in a system where users of the system *all* run SPV nodes and only miners run full nodes.
- Give you the same level of privacy. A lightweight node must ask network nodes for specific blocks or transactions. Asking for those blocks/transactions can reveal information about your keys and txs.
    - Improved slightly by bloom filters
    - Improved further by client-side filtering (BIP 1578/158)
- Detect false negatives. Network nodes can lie-by-omission (ie not tell you about transactions in blocks).

A note on “Satoshi’s Vision”

- Satoshi believed that eventually, most users would run light nodes: “*If the network becomes very large, like over 100,000 nodes, this is what we'll use to allow common users to do transactions without being full blown nodes. At that stage, most users should start running client-only software and only the specialist server farms keep running full network nodes, kind of like how the usenet network has consolidated.*”. (https://bitcointalk.org/index.php?topic=125.msg1149#msg1149)
- However, the whitepaper states that “*Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.”*

Bloom Filters

- Defined in BIP 37 (https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki).
- Implemented in Bitcoin Core in August 2012 (https://github.com/bitcoin/bitcoin/pull/1795/files)
- 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 (https://jonasnick.github.io/blog/2015/02/12/privacy-in-bitcoinj/) (https://eprint.iacr.org/2014/763.pdf)
- Places load on the server. Can be used as to DOS servers providing the filters (https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-May/012636.html)
- Doesn’t work with segwit (https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-April/015894.html)
- Generally not advised. Will probably be disabled by default in the next Bitcoin Core version (https://github.com/bitcoin/bitcoin/pull/16152).

Compact Block Filters

- Defined in BIPs 157/158 (https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki)
- 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 (initially in roasbeef fork) (https://github.com/Roasbeef/btcd/pull/8)
- Compact Block Filters (https://github.com/bitcoin/bitcoin/pull/12254) and index (https://github.com/bitcoin/bitcoin/pull/14121) are implemented for Bitcoin Core. P2P implementation is WIP.

Checkpoints

Origin of checkpoints

- Satoshi added in 0.3.2 to prevent 51% attack from reorging the blockchain too far (200 blocks prior to release) (https://github.com/bitcoin/bitcoin/commit/4110f33cded01bde5f01a6312248fa6fdd14cc76#diff-118fcbaaba162ba17933c7893247df3aR1344)
- Three checkpoints added:
  - 11111
  - 33333
  - 68555
- *The security safeguard makes it so even if someone does have more than 50% of the network's CPU power, they can't try to go back and redo the block chain before yesterday. (if you have this update)* (https://bitcointalk.org/index.php?topic=437.msg3807#msg3807)
- What it actually did: you can't extend a chain block 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. (https://github.com/bitcoin/bitcoin/pull/492). There was concern about this from Pieter and Greg. (https://github.com/bitcoin/bitcoin/pull/492#issuecomment-2007585)
  - That was later changed to only skip script validation for ancestors of the checkpointed block (https://github.com/bitcoin/bitcoin/pull/5927)
- Then checkpoints also started to be used as an anti-DoS measure; we'd not process block headers that were timestamped older than the last checkpointed time. (https://github.com/bitcoin/bitcoin/pull/534, https://github.com/bitcoin/bitcoin/pull/534/files#diff-7ec3c68a81efff79b6ca22ac1f1eabbaR1365)
- For many years more checkpoints were added:

Jul 2010	-	height 70567
Sep 2010	-	height 74000
Jan 2011	-	height 105000
Apr 2011	-	height 118000
July 2011	-	height 134444
Aug 2011	-	height 140700
Feb 2012	-	height 168000

    - Aug 2012 - height 193000
    - Dec 2012 - height 210000
    - Jan 2013 - height 216116
    - Mar 2013 - height 225430
    - Aug 2013 - height 250000
    - Jan 2014 - height 279000
    - Jul 2014 - height 295000

Evolution of thinking

- Stopped adding checkpoints at 295,000. (https://github.com/bitcoin/bitcoin/pull/4541)
- Generally, there's concern philosophically about developer centralization/control if we keep pushing out new checkpoints. That final checkpoint was opposed by Peter Todd and Pieter Wuille (https://github.com/bitcoin/bitcoin/pull/4541#issuecomment-49469837)
- Separated checkpoints from signature skipping during IBD in 0.14, replaced with `-assumevalid`. (https://github.com/bitcoin/bitcoin/pull/9484, https://github.com/bitcoin/bitcoin/pull/9484/files#diff-24efdb00bfbe56b140fb006b562cc70bR1724)
- Checkpoints still lock in the chain as a defense against low-difficulty headers being used to overwhelm a node.
- Started to use a new metric, nMinimumChainWork, for more anti-DoS protections, in places where checkpoints used to be used. (https://github.com/bitcoin/bitcoin/pull/9053, https://github.com/bitcoin/bitcoin/pull/9053/files#diff-7ec3c68a81efff79b6ca22ac1f1eabbaL1743)
    - *IBD check uses minimumchain work instead of checkpoints.*

      This introduces a 'minimum chain work' chainparam which is intended to be the known amount of work in the chain for the network at the time of software release. If you don't have this much work, you're not yet caught up.

      This is used instead of the count of blocks test from checkpoints.

      This criteria is trivial to keep updated as there is no element of subjectivity, trust, or position dependence to it. It is also a more reliable metric of sync status than a block count.

Going forward

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

- Need a defense against low-diff headers (memory blowup).
- One approach -- soft fork in a min-difficulty that is higher than current. This kind of sucks though, because it is consensus-level.

- Another approach -- p2p only changes. Can be done with new p2p messages, but isn't a priority.
- assumeutxo might make this moot - the utxo sync point becomes an effective checkpoint, maybe.

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.

- Dave Harding

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 values could come from several places:
  - Hardcoded into the source code and reviewed as part of the release cycle
  - provided as config option or RPC input
  - 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

Proposal (https://github.com/jamesob/assumeutxo-docs/tree/2019-04-proposal/proposal)

- Is this a change in the trust model?

notes:

- Would this lead to trusting developers?
- Anyone can theoretically review the UTXO set, by sync'ing from genesis. It's much harder to review the code (and toolchain!)
- Does this 'kick the can down the road' in terms of making IBD sustainable? Will everyone eventually just use this and only a very few groups generate the snapshot?
- Would adding this to the coinbase commitment just make everyone trust the miners?
- Can I just give someone a bad UTXO set and magic coins out of thin air? No, because the commitment value is hard-coded.

Committed UTXO hashes

Basic idea is to offer a new security model with lower cost than running a full node.

We're going to look at 4 different UTXO proposals:

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

All give different security models.

Alan Reiner's 'trust-free lite nodes'

- Proposed by Alan Reiner in a bitcointalk post here: https://bitcointalk.org/index.php?topic=88208.0
- 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.
- 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: *"An alt-"chain" is probably the wrong way to think of this. It's committed data. There*

doesn't need to be a chain. Each bitcoin block could have 0 or 1 tree commitments of a given type." (gmaxwell - https://bitcointalk.org/index.php?topic=88208.msg1064165#msg1064165)
- Pieter had implemented ultraprune (https://github.com/bitcoin/bitcoin/pull/1677) at roughly the same time. Pieter suggested that Ultraprune could be thought as a step towards these ideas (https://bitcointalk.org/index.php?topic=88208.msg1316642#msg1316642

Peter Todd's TXO MMR commitments

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-May/012715.html
- A new hash root is committed to in the block. The structure is a Merkle Mountain Range - a type of deterministic, indexable, insertion ordered merkle tree
- 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 log2(n) sized proofs consisting of a merkle path to the tip of the tree.
- At an extreme, with TXO commitments we could even have no UTXO set at all. Transactions would simply be accompanied by TXO commitment proofs showing that the outputs they wanted to spend were still unspent
- A more realistic implementation is to have a UTXO cache for recent transactions, with TXO commitments acting as an alternate for the (rare) event that an old txout needs to be spent.
- The spender provides the proof that the TXO exists and is unspent. Proofs can be generated and added to transactions without the involvement of the signers, even after the fact; there's no need for the proof itself to signed and the proof is not part of the transaction hash.
- 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.
- A later post https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-February/013591.html 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 that are unlikely to be spent. In the event those UTXO's are spent, transactions and blocks spending them can trustlessly provide the necessary data to temporarily fill-in the node's local TXO set database, allowing the next commitment to be calculated.
    - By not committing the TXO commitment in the block itself, we obsolete the requirement for delayed TXO commitments.

    - 1. Implement a TXO commitment scheme with the ability to efficiently store the last n versions of the commitment state for the purpose of reorgs (a reference-counted scheme naturally does this).
    - 2. Add P2P support for advertising to peers what parts of the TXO set you've pruned.
    - 3. Add P2P support to produce, consume, and update TXO unspentness proofs as part of transaction and block relaying.

Bram Cohen's TXO bitfield

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-March/013928.html
- https://www.youtube.com/watch?v=52FVkHlCh7Y
- http://diyhpl.us/wiki/transcripts/sf-bitcoin-meetup/2017-07-08-bram-cohen-merkle-sets/
- 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.

Pieter Wuille’s Rolling UTXO set hashes

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014337.html
- Easy to come up with a system when efficient updating is not required, for example, concatenate all the UTXOs in a canonical order and then hash.
- Efficiency requires that we be able to hash the data in any arbitrary order, and remove as well as add elements.
- Proposal is initially not for committing data. Use cases are:
    - Replacement for Bitcoin Core's gettxoutsetinfo RPC's hash computation. This currently requires minutes of I/O and CPU, as it serializes and hashes the entire UTXO set. A rolling set hash would make this instant, making the whole RPC much more usable for sanity checking.
    - Assisting in implementation of fast sync methods with known good blocks/UTXO sets.
    - Database consistency checking: by remembering the UTXO set hash of the past few blocks (computed on the fly), a consistency check can be done that recomputes it based on the database.
- Two proposed solutions:
    - 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 (https://code.fb.com/security/homomorphic-hashing/)

Rolling hash of set of objects

- Bellare-Micciancio paper “A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost” https://cseweb.ucsd.edu/~mihir/papers/inchash.pdf suggests ways of incremental hashing:
    - XHASH - hash individual objects and XOR - is trivially insecure under Wagner’s attack (https://link.springer.com/content/pdf/10.1007%2F3-540-45708-9_19.pdf)
    - AdHASH - hash individual objects and add modulo some large prime - is also insecure under Wagner’s attack. N-bit hash length gives O(2^(2*sqrt(n)-1)) security, so a 256 bit hash gives 31 bits of security. A ~400000-bit hash is needed to give 128 bits of security if repetition is allowed (we don’t need to allow repetition for the UTXO set, but it’s nice if verification software doesn’t need to check for duplicates)
    - MuHASH - hash individual objects and multiply modulo some large prime - is secure under the DL assumption. If it can be attacked efficiently, then the same method can be used to attack the DL problem. A 3072-bit hash gives about 128 bits of security (and the output can be hashed to a 256 bit digest with no loss of security)
- If we use MuHash, we remove an item by finding the inverse of its hash under multiplication modulo the prime. This is quite expensive, so in fact we just store the numerator (all TXOs) and denominator (STXOs) and only do one inverse to calculate the rolling hash.

Hashing onto a curve

- Elliptic Curve Multiset Hash (https://arxiv.org/pdf/1601.06502.pdf) 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 216.
- Properties:
    - Set homomorphic
    - Collision resistant

All methods are perfectly parallelizable (since ordering is not important)

Further reading

- UHS - Full-node security without maintaining a full UTXO set https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-May/015967.html
- utreexo - A dynamic accumulator for Bitcoin state https://dci.mit.edu/research/2018/11/28/utreexo-a-dynamic-accumulator-for-bitcoin-state-a-description-of-research-by-thaddeus-dryja
- Accumulators - a scalablehttps://dci.mit.edu/research/2018/11/28/utreexo-a-dynamic-accumulator-for-bitcoin-state-a-description-of-research-by-thaddeus-dryja drop-in for Merkle Trees https://diyhpl.us/wiki/transcripts/scalingbitcoin/tokyo-2018/accumulators/ , https://www.youtube.com/watch?v=IMzLa9B1_3E&feature=youtu.be&t=3522
- Flyclient - Super-Light Clients for Cryptocurrencies https://eprint.iacr.org/2019/226