Future Technological Directions in Bitcoin

Introduction

This is a document on coming bitcoin tech improvements and research. It does not cover non-bitcoin projects (eg. Mimblewimble, ZCash, Ethereum) though much which applies here could be applied to those systems too.

When looking at Bitcoin tech improvements, we can divide them into several rough categories:

1. Implementation improvements, ie. better code.
2. Peer protocol improvements, ie. better communications between nodes.
3. Consensus protocol improvements, ie. backwards-compatible upgrades to Bitcoin's blockchain itself aka "soft forks".
4. Fork consensus issues, ie. new upgrade methods, and non-backwards-compatible changes aka "hard forks".
5. Things built on top of bitcoin, ie. "Layer 2" protocols and projects.

These are not mutually exclusive: some peer protocol improvements would be more effective with consensus changes, and some layer 2 uses require consensus changes. Where possible, I cover the consensus idea before its usage.

Background: Bitcoin

Bitcoin is a ledger of inputs and outputs. Each output says how it can be spent, using a stack-based scripting language with cryptographic primitives (eg. OP_CHECKSIG). The classic output script is a simple "<key> OP_CHECKSIG" which would be spent with the input "<signature>" which signs the spending transaction with the given key.

Modern practice is not to place the output script itself in the output (where it has to be remembered until it's spent) but place a hash of the script, so-called Pay-to-Script-Hash, and have the actual script be the last item in the input. That takes more space for the moment when it's being spent, but less before it's spent, and potentially less after it's validated (as it can be

discarded). The set of unspent transaction outputs is called the UTXO set, and keeping it small is considered a critical long-term scaling issue as it needs to be constantly looked up for validation.

Consensus Protocol Improvements

Segregated Witness

Status: Deployed
References:

- https://bitcoincore.org/en/2016/06/24/segwit-next-steps/
- https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki
- https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki

I won't go into the details of this significant simplification, but the important change is that Bitcoin scripts are now versioned. Only version 0 is defined so far (basically identical to classic Bitcoin scripting), but future versions can be completely different if we want. All current upgrades involve changing one of the NOP opcodes, but they have to be changed to opcodes of simple form "fail unless this condition is true", eg OP_CHECKSEQUENCEVERIFY. They can't consume or add to the stack, since that would allow creating scripts which would fail previously, and will now pass: that's a backwards-incompatible change!

For some changes, this restriction doesn't matter, so I expect NOP-upgrades to continue until there's a compelling reason to introduce v1. But the door is now open when we need it.

SEGWIT BECH32 ADDRESSES

- https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki
- http://bitcoin.sipa.be/bech32/demo/demo.html
- https://www.youtube.com/watch?v=NqiN9VFE4CU
- https://github.com/sipa/ezbase32/blob/master/dist32.cpp#L13L25

These look like bc1q4xwnullywullewuelewuulywulewuluhwuelwu.

30 bit code detects up to 4 errors in 71 characters (max addr len), or 5 "common substitution" errors. Equivalent detection power to old 32-bit truncates SHA at 3.53 per address, for "common substitutions", 4.85 per address. At 2.4% error rate, we'd expect one per P2WPKH address, and we'd need 5 or more to risk an error being accepted, which happens about 1 in 2^39. But the BCH constants were chosen so that single-bit errors are detected up to 5 in 71

characters, and encoding such that q/p r/t z/2 e/a s/5 4/h y/v x/k etc are 1 bit apart. This means for common substitutions, we need 6 errors to risk it passing, which happens about 1 in 2^42.

MAST

- https://www.mail-archive.com/bitcoin-dev@lists.linuxfoundation.org/msg05991.html
- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-September/014963.html
- http://diyhpl.us/wiki/transcripts/bitcoin-core-dev-tech/2017-09-07-merkleized-abstract-syntax-trees/
- https://github.com/jl2012/bips/blob/vault/bip-0114.mediawiki
- https://github.com/bitcoin/bips/blob/master/bip-0116.mediawiki
- https://github.com/bitcoin/bips/blob/master/bip-0117.mediawiki

Merkelized Abstract Syntax Trees are a general concept: when bitcoin developers talk about it, they're talking about reworking bitcoin scripts into a series of "OR" branches, and instead of the output committing to the whole script, you commit to the head of the tree. To spend it, you only need to provide the branch of the script you're using, and the hashes of the other branches. This can improve privacy, and also shrink the total size of large scripts, particularly if there's a short, common case, and a long, complex rare case. Note that each key is 33 bytes and each signature about 72 bytes, and each merkle branch only 32 bytes.

Johnson Lau's proposal (BIP 114) is that basically v1 segregated witness scripts would be MAST. A MAST version is added underneath that, for future expansion. The only non-triviality is that the script can be divided into multiple parts, and hashed into the tree as H(H(part1)|H(part2)|H(part3)...). This is to enable hidden clauses in a safe way: if two parties are signing the transaction, one might want to append more conditions to its branch, and can just provide the hash to the other party.

Mark Friedenbach's proposal is in two parts. BIP 116 proposes a new MERKLEBRANCHVERIFY op, allowing efficient checking that a stack element is inside a merkle tree. This is a new form of optimized merkle tree.

BIP 117 adds the semantic that extra elements on the stack at the end execution are actually a script (or number of scripts to be pasted together) which should be executed. This means a MAST script would use MERKLEBRANCHVERIFY to check that the handed-in blob was in the tree, then leave it on the stack for execution. This is quite neat. However, in its presented form, it's not statically analyzable, nor does it enforce opcode or signature op limits. And unfortunately, you can't have extra values on the stack in segwit v0, so the proposal allows use of the rarely-used altstack as well.

Taproot

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-January/015614.html

Greg Maxwell points out that MAST is often, if not always, a choice between a “normal” case which (with Signature Aggregation: see below) comes down to a pay-to-pubkey-hash, and a more complicated exceptional case. Unfortunately, the existence of unused branches still distinguishes MAST spends from other simple payments.

Fortunately, Taproot shows it’s possible to have a MAST which looks like a normal P2WPKH. This would probably be a new segwit version, so everything looks the same. The new pay-to-pubkey hash would be spendable with a key (P) and signature as normal, but *also* spendable by a base key (C) and a script, where P = C + H(C | script). This works because you must have deliberately derived P from C and the script: there’s no way to reverse this hash, so if P really was a normally-generated key, nobody can turn it into a script-style spend.

Signature aggregation

- http://diyhpl.us/wiki/transcripts/bitcoin-core-dev-tech/2017-09-06-signature-aggregation/

Bitcoin’s current secp256k1 ECDSA signature scheme is not optimal: if you want to sign a transaction with two keys, you need two signatures. There’s a simpler signature system (sometimes called Schnorr signatures) which has nicer properties, unfortunately there’s no standard for such signatures.
Key aggregation: would allow N parties to cooperate to produce a new single key, which they could cooperate later to sign things with. This saves space and increases privacy.
Unfortunately, doing this naively is fraught: you can select one key to cancel out another. The accepted solution to this is Bellare-Neven multisignatures.
Signature aggregation: one signature, for multiple keys. Parties only need to come together to produce signature. This lets us have one signature for the entire transaction, even though each input is signed by a separate key; the new “OP_CHECKSIG” would always succeed, but stash the key somewhere. These would all be validated against the aggregated signature at the end of the transaction processing. This gives significant advantage to combining transactions, incentivizing coinjoin-style operations (which can be done trustlessly).
Batch validation: It turns out you can validate 1000 independent signatures together faster than you can validate 1000 signatures separately, though it only tells you whether they’re all good, or something is wrong. But this is perfect for historic block validation, where we can check the entire set of transactions at once.

Scriptless Scripts

- https://www.youtube.com/watch?v=3pd6xHjLbhs&feature=youtu.be&t=1h35m58s
- https://scalingbitcoin.org/transcript/stanford2017/using-the-chain-for-what-chains-are-good-for

There's a method for trustless exchange between separate blockchains called "atomic swaps" (Teir Nolan). Alice agrees to trade 1 ABC for 1 of Bob's XYZ; Alice provides a hash, and creates a ABC transaction which requires Bob's sig and the preimage of that hash. Now Bob creates an ABC transaction which requires Alice's sig and the preimage of that hash. Alice collects Bob's XYZ output, revealing the preimage, and now Bob can collect Alice's ABC.

(This same method is used to make Lightning network transactions atomic across multiple parties. Also, timeouts are required above in case Alice or Bob flakes out).

With Schnorr Signatures, Andrew Poelstra shows that you don't need preimages: we can set up transactions such that by Alice signing the transaction to collect Bob's transaction, Bob can generate a signature to collect Alice's transaction. Not only does this save space, but these transactions are uncorrelatable, and look like any other payment.

Simplicity

- https://blockstream.com/simplicity.pdf
- https://bitcoinmagazine.com/articles/introducing-programming-language-so-simple-it-fits-t-shirt/

"Simplicity is a typed, combinator-based, functional language without
loops and recursion, designed to be used for crypto-currencies and blockchain applications. It aims to improve upon existing cryptocurrency~~crypto-currency~~ languages, such as Bitcoin Script and Ethereum's EVM, while avoiding some of the problems they face. Simplicity comes with formal denotational
semantics defined in Coq, a popular, general purpose software proof assistant. Simplicity also includes operational semantics that are defined with an abstract machine that we call the Bit Machine. The Bit Machine is used as a tool for measuring the computational space and time resources needed to evaluate Simplicity programs. Owing to its Turing incompleteness, Simplicity is amenable to static analysis that can be used to derive upper bounds on the computational resources needed, prior to execution. While Turing incomplete, Simplicity can express any finitary function, which we believe is enough to build useful "smart contracts" for blockchain applications."

Covenants

- http://fc16.ifca.ai/bitcoin/papers/MES16.pdf
- http://diyhpl.us/wiki/transcripts/scalingbitcoin/milan/covenants/
- https://github.com/jl2012/bips/blob/vault/bip-0ZZZ.mediawiki

Covenants are a general method of introspection of a transaction; a script can already check sequence number (OP_CHECKSEQUENCEVERIFY) and locktime number (OP_CHECKLOCKTIMEVERIFY) in the spending transaction. The initial Mosel, Eyal, Sirer paper suggested OP_CHECKOUTPUTVERIFY, to make sure the spender was in turn spending the funds in some way. They used this to create a "Vault", where coins can only be spent by a transaction which in turn can only be spent after a day, or with a separate emergency key. The idea is that an attacker who compromises your private key (but not your emergency key!) gives you a day to stop them.

This was (almost accidentally) enabled in Blockstream's Elements Alpha sidechain, which added OP_CHECKSIGFROMSTACK; Russell O'Connor used this to allow almost arbitrary properties of a transaction to be verified. It builds the transaction on the stack, then checks that the signature you use to sign the transaction matches. More realistically, Johnson Lau proposed a OP_PUSHTXDATA which can push several types of data from the current transaction onto the stack. This is only possible with a new version of the Segwit Script (currently only version 0 is defined), as backwards compatible opcodes must not alter the stack.

Confidential Transactions

- https://people.xiph.org/~greg/confidential_values.txt
- https://blockstream.com/bitcoin17-final41.pdf
- https://eprint.iacr.org/2017/1066.pdf
- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-December/015346.html

Confidential Transactions are a cryptographic method of concealing the amount of an output, but in such a way that everyone can see that the outputs - inputs - fee is equal to zero. This avoids many of the simpler approaches to blockchain analytics which compromise privacy. Unfortunately, this has never been proposed for inclusion into bitcoin because of the size of the values: about 2.5k each! They are also fairly slow to validate (14/10ms)

A more recent paper by Bunz et. al. called "bullet proofs" drops this significantly: they're about 4ms to check, and scale up super-linearly if you batch them. More importantly, they're only 674 bytes each (for a 64-bit range proof, which is actually more than we need), but can be combined across a transaction in an Nlog(N) manner: 2 outputs is only 738 bytes, 8 is 866 bytes.

Note that a normal signature takes only 0.084 msec to validate, so there's still a significant performance hit. But further refinements may bring this into the realm of possibility; whether the bitcoin community wants to use some of its precious limited block size for this remains to be seen though.

Client-side validation

- Eg. http://diyhpl.us/wiki/transcripts/scalingbitcoin/milan/client-side-validation/ https://scalingbitcoin.org/milan2016/presentations/D2%20-%20A%20-%20Peter%20Todd.pdf

This is more handwaving than most things on the list, in that there's a ton of work to make incentives correct. But the base idea is to consider any sufficiently-POW block to be valid, and have the spender provide a proof that the chain of coins they're relying on is correct. This shifts the burden, but potentially allows sharding and other mining scale. Having such multi-megabyte proofs is burdensome, so probabilistic techniques could be used (say, using the blockchain itself as a beacon) to ensure that cheating is unprofitable.

UTXO commitments

- Andrew Miller's proposal and implementation
- Alan Reiner's proposal
- Mark Friedenbach worked on one
- Peter Todd has an implementation
- https://github.com/bramcohen/MerkleSet

There's a very old idea in bitcoin, that each block should commit not just to the transactions within it, but the full set of unspent transaction outputs (UTXOs). This allows several improvements:

- You can provide a merkle proof that an option is unspent, not just that it exists.
- Similarly, if the tree is ordered canonically, you can prove that it *has* been spent by proving the two adjacent UTXOs where this UTXO would be found.
- 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 general approach is difficult, however: the utxo set is large and growing, and requiring nodes to validate it is a significant burden: once committed to, the format will be permanent, so it requires careful design.
A naive approach of simply using a merkle tree ordered by txid would be horrible to update; much more expensive than block validation today; there are over 60M unspent outputs. Ordering by block order would be nicer to add, but doesn't help to remove and requires double-proofs.

Bram Cohen presented an efficient design of Merkle Patricia Tries, which involves multiple clever design choices, but is non-trivial. The exact numbers for a full implementation are still unknown for current UTXO sizes (his implementation is in Python).

UTXO Proofs

UTXO sets even allow a kind of inversion, where wallets provide proofs that their transaction inputs are in the utxo set, and miners just validate those and update their utxo set accordingly. This shifts the burden of UTXO management from the miners to the wallets. Hybrid schemes (where this only applied to older UTXOs) are also possible.

TXO commitments

- https://petertodd.org/2016/delayed-txo-commitments#slow-path-calculating-pending-txo-commitments

Peter Todd suggests a delayed commitment, with a full TXO set current for some N blocks back, and a “update” set of spent STXOs since then. This works as the canonical set is not latency critical because you have N blocks to generate it, and the STXO set is small (in the tens of thousands, depending on N). Nodes only need a handful of “peaks” in the TXO tree so they can append to it; to spend an input from the old set (TXO set) you’d need to provide a proof that it was in the set, which also gives enough information to mark the TXO spent in the set. This proof would be under 1k at the moment (26-deep tree, 32 byte hashes, plus the tx outpoint and script), but that would still a significant tx size increase, trading off per-tx-bandwidth for per-utxo-storage. Unfortunately, this means the wallet needs to track what’s happening to update its proofs of position.

Fraud proofs

- https://en.bitcoin.it/wiki/User:Gmaxwell/features#Proofs
- Problems w/ bip180 https://github.com/bitcoin/bips/blob/master/bip-0180.mediawiki

The original Satoshi paper suggested lite nodes which only kept the 80-byte bitcoin headers “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.”. Unfortunately, that’s both open to abuse which forces them to validate every block, and insufficient, since a block cannot be validated in isolation.

What’s wanted is a way to compactly prove every possible violation in a block. You can compactly prove any transaction in the block, using merkle proofs, but not all of violations are

similarly provable. You can prove “that input has already been spent” by providing the previous transaction which spent it. If each transaction input commits to where the output was in the blockchain, you can also prove “that input doesn’t exist”, otherwise you can use UTXO commitments.

You also can’t prove “block is oversize” or “block claims excessive fees” compactly, but these can be solved by eg “Replace the hash tree with one that H((fees_left+fees_right)||H(left)||H(right))”. Similarly for block size, commit to the total size.

Unfortunately, neither “merkle root is garbage” nor “UTXO commitment is garbage” can be compactly proven: for that you’d seem to need all the transactions. And in a world where everyone relies on fraud proofs, particularly if miners can mine without having complete block knowledge, miners can simply withhold the data to avoid being disproven. Greg Maxwell has suggested forcing publication using a fountain-code style protocol, but it would be very different from the current model and would require more research.

Peer Protocol Improvements

TXO bitfields

- https://www.youtube.com/watch?v=52FVkHlCh7Y
- http://diyhpl.us/wiki/transcripts/sf-bitcoin-meetup/2017-07-08-bram-cohen-merkle-sets/
- https://youtu.be/52FVkHlCh7Y?t=2292 (Greg Maxwell’s 2 minute summary)

Bram uses a similar approach with TXO bitfields. 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 unlike the TXO set in Peter Todd’s commitment, is immutable. Nodes then 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, which is 783,038,273 as of block 502340, vs UTXO set of 62,028,853, 12.6x the UTXO set, so still 1/20th of the size. A less-naive implementation could be more compact, given the bitfield will be sparse. Wallets simply send their proof of position for each input. Moreover, this is not a block consensus change, it’s merely a peer-to-peer upgrade.

Rolling UTXO Set Hashes

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014337.html

A completely different approach, which is somewhat orthogonal, is Pieter Wuille's approach of a rolling (256 bit) "UTXO set hash that is very efficient to update, but does not support any compact proofs of existence or non-existence.". This would be useful for the "prove I have the right UTXO set" both for internal validation, and for less-than-full nodes which have a canned UTXO set from somewhere (or use it while bootstrapping).

It turns out that naively XORing hashes together to give a set hash is insecure: "Gaussian elimination can easily find a subset of random hashes that XOR to a given value".

Two approaches which are secure give numbers as follows:

- Computing the hash from just the UTXO set takes (1) 2m15s (2) 9m20s
- Processing all creations and spends in an average block takes (1) 24ms (2) 100ms
- Processing precomputed per-transaction aggregates in an average block takes (1) 3ms (2) 0.5ms

Pieter prefers (1) since it's simpler and faster.

Dandelion

- https://arxiv.org/abs/1701.04439
- https://github.com/sbaks0820/bitcoin-dandelion/issues/1

Dandelion is a privacy improvement for submitting transactions by Venkatakrishnan, Fanti and Viswanath at University of Illinois. There are numerous companies who actively try to deanonymize bitcoin transactions by connecting to all the nodes they can reach and trying to trace the origin of each transaction, for correlation against others. Dandelion's idea is simple: each transaction when it arrives is advertised to a single host with 90% probability, or more widely with a 10% probability (or if it's not seen again for some period of time, in case it got lost). These are called the stem and fluff stages respectively. This makes it very hard for an attacker to determine the original IP address of the transaction.

Neutrino (BIP 157/158)

- https://youtube.com/watch?v=7FWKc8lM4Ek
- https://github.com/Roasbeef/bips/blob/master/gcs_light_client.mediawiki
- https://github.com/lightninglabs/neutrino

Neutrino is a better protocol for lightweight clients. Current SPV wallets send a bloom filter to the full nodes they connect to, and the bloom filter contains the pubkey and pubkey hash entries, so when one is seen, it's fairly easy to determine if it belongs to a wallet.

Neutrino inverts this: nodes send a Golomb-Rice coded set (a more optimal approach, but similar in spirit, to a bloom filter) which summarizes the block; averaging 20k. If the lite client thinks something is interesting, it then downloads the full block, possibly from a different peer. This proposal does not go all the way to having the block itself commit to the set (though that could be soft-forked in later), but does use a chain of headers where each commits to the previous, so that any lying node is easily caught.

Transaction compression

- Advances in Block Propagation, Greg Maxwell
  https://www.youtube.com/watch?v=EHIuuKCm53o
  https://people.xiph.org/~greg/gmaxwell-sf-prop-2017.pdf

A specialized compressor can compress individual transactions (no cheating by using context!) by about 28% on average over the entire blockchain. The CPU costs of doing so are yet to be measured properly.

Set reconciliation for transaction relay

- As above

Chatting about what transactions we have actually consumes more bandwidth than blocks; this could be reduced by having a low fanout (1.5 peers) and then using periodic set reconciliation to catch missing ones.

Block Template Delta Compression

- As above

BIP 152 (Compact Block Relay) can operate in high bandwidth mode (just send the block) or non-high-bandwidth mode (send the INV message). Core defaults to HB mode for the last 3 peers which gave it a block (40kb isn't that expensive).

Big cost in set reconstruction is permutations; sending a template every 30 seconds, with the AV1 range encoder gives average 3k per block. Adds back state, but not too bad if we only do it with the 3 HB peers.

NODE_NETWORK_LIMITED

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014314.html

NODE_NETWORK_LIMITED_LOW (288 blocks) and NODE_NETWORK_LIMITED_HIGH (1152 blocks).

Currently pruned nodes don't advertise NODE_NETWORK in their service bits, so they don't serve any blocks. Yet recent blocks are the most in-demand; the last block, and the recent ones (when a node comes back online). With neutrino, block downloads may also increase. These allow pruned nodes to share the load.

Peer authentication and peer encryption (bip150 and bip151)

- https://github.com/bitcoin/bips/blob/master/bip-0150.mediawiki
- https://github.com/bitcoin/bips/blob/master/bip-0151.mediawiki
- http://diyhpl.us/wiki/transcripts/sf-bitcoin-meetup/2017-09-04-jonas-schenlli-bip150-bip151/

Currently, the bitcoin gossip protocol is both unencrypted and unauthenticated. This makes it easier for passive observers to determine exact transaction transits, as well as giving them access to existing information leakage (eg. bloom filters).

Also, if you are running a lite node, authentication lets you make sure you're connected to your trusted full node.

TODO section

Forks

- BIP 8 (4)
    - https://github.com/bitcoin/bips/blob/master/bip-0008.mediawiki
- Additive PoW soft-forks
- HF options (4)
    - FIXME: details of spoonnet and spoonnet2
    - https://bitcoinhardforkresearch.github.io/

Implementation Improvements

- Better fee estimation (estimatesmartfee and future directions, branch-and-bound, etc.)

Coinjoin/Zerolink/Coinshuffle++

Layer 2

- Zero-knowledge contingent payments
    - https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement
- Lightning
    - https://github.com/lightningnetwork/lightning-rfc
    - Watchtowers (2)
    - Decorrelation (3)
    - FIXME: select from:
        - https://github.com/lightningnetwork/lightning-rfc/wiki/Brainstorming
- amortized secure multiparty computation (4)
    - https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/instantaneous.pdf
- Discreet (sic) log contracts
    - https://adiabat.github.io/dlc.pdf
    - http://diyhpl.us/wiki/transcripts/discreet-log-contracts/
- Sidechains
- Joinmarket, tumblebit, coinshuffle, etc.
- Atomic swaps via some marketplace