```
BIP: 157
Layer: Peer Services
Title: Client Side Block Filtering
Author: Olaoluwa Osuntokun <laolu32@gmail.com>
        Alex Akselrod <alex@akselrod.org>
        Jim Posen <jimpo@coinbase.com>
Comments-Summary: None yet
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0157
Status: Draft
Type: Standards Track
Created: 2017-05-24
License: CC0-1.0
```

Abstract

This BIP describes a new light client protocol in Bitcoin that improves upon currently available options. The standard light client protocol in use today, defined in BIP 37[1], has known flaws that weaken the security and privacy of clients and allow denial-of-service attack vectors on full nodes[2]. The new protocol overcomes these issues by allowing light clients to obtain compact probabilistic filters of block content from full nodes and download full blocks if the filter matches relevant data.

New P2P messages empower light clients to securely sync the blockchain without relying on a trusted source. This BIP also defines a filter header, which serves as a commitment to all filters for previous blocks and provides the ability to efficiently detect malicious or faulty peers serving invalid filters. The resulting protocol guarantees that light clients with at least one honest peer are able to identify the correct block filters.

Motivation

Bitcoin light clients allow applications to read relevant transactions from the blockchain without incurring the full cost of downloading and validating all data. Such applications seek to simultaneously minimize the trust in peers and the amount of bandwidth, storage space, and computation required. They achieve this by downloading all block headers, verifying the proofs of work, and following the longest proof-of-work chain. Since block headers are a fixed 80-bytes and are generated every 10 minutes on average, the bandwidth required to sync the block headers is minimal. Light clients then download only the blockchain data relevant to them directly from peers and validate inclusion in the header chain. Though clients do not check the validity of all blocks in the longest proof-of-work chain, they rely on miner incentives for security.

[1]https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki
[2]https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-May/012636.html

BIP 37 is currently the most widely used light client execution mode for Bitcoin. With BIP 37, a client sends a Bloom filter it wants to watch to a full node peer, then receives notifications for each new transaction or block that matches the filter. The client then requests relevant transactions from the peer along with Merkle proofs of inclusion in the blocks containing them, which are verified against the block headers. The Bloom filters match data such as client addresses and unspent outputs, and the filter size must be carefully tuned to balance the false positive rate with the amount of information leaked to peer. It has been shown, however, that most implementations available offer virtually *zero privacy* to wallets and other applications[3][4]. Additionally, malicious full nodes serving light clients can omit critical data with little risk of detection, which is unacceptable for some applications (such as Lightning Network clients) that must respond to certain on-chain events. Finally, honest nodes servicing BIP 37 light clients may incur significant I/O and CPU resource usage due to maliciously crafted Bloom filters, creating a denial-of-service (DoS) vector and disincentizing node operators from supporting the protocol[5].

The alternative detailed in this document can be seen as the opposite of BIP 37: instead of the client sending a filter to a full node peer, full nodes generate deterministic filters on block data that are served to the client. A light client can then download an entire block if the filter matches the data it is watching for. Since filters are deterministic, they only need to be constructed once and stored on disk, whenever a new block is connected to the chain. This keeps the computation required to serve filters minimal, and eliminates the I/O asymmetry that makes BIP 37 enabled nodes vulnerable. Clients also get better assurance of seeing all relevant transactions because they can check the validity of filters received from peers more easily than they can check completeness of filtered blocks. Finally, client privacy is improved because blocks can be downloaded from *any source*, so that no one peer gets complete information on the data required by a client. Extremely privacy conscious light clients may opt to anonymously fetch blocks using advanced techniques such a Private Information Retrieval[6].

Definitions

`[]byte` represents a vector of bytes.

`[N]byte` represents a fixed-size byte array with length N.

CompactSize is a compact encoding of unsigned integers used in the Bitcoin P2P protocol.

double-SHA256 is a hash algorithm defined by two invocations of SHA-256: `double-SHA256(x) = SHA256(SHA256(x))`.

[3] https://eprint.iacr.org/2014/763.pdf
[4] https://jonasnick.github.io/blog/2015/02/12/privacy-in-bitcoinj/
[5] https://github.com/bitcoin/bips/blob/master/bip-0111.mediawiki
[6] https://en.wikipedia.org/wiki/Private_information_retrieval

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Specification

Filter Types

For the sake of future extensibility and reducing filter sizes, there are multiple *filter types* that determine which data is included in a block filter as well as the method of filter construction/querying. In this model, full nodes generate one filter per block per filter type supported.

Each type is identified by a one byte code, and specifies the contents and serialization format of the filter. A full node MAY signal support for particular filter types using service bits. The initial filter types are defined separately in BIP 158, and one service bit is allocated to signal support for them.

Filter Headers

This proposal draws inspiration from the headers-first mechanism that Bitcoin nodes use to sync the block chain[7]. Similar to how block headers have a Merkle commitment to all transaction data in the block, we define filter headers that have commitments to the block filters. Also like block headers, filter headers each have a commitment to the preceding one. Before downloading the block filters themselves, a light client can download all filter headers for the current block chain and use them to verify the authenticity of the filters. If the filter header chains differ between multiple peers, the client can identify the point where they diverge, then download the full block and compute the correct filter, thus identifying which peer is faulty.

The canonical hash of a block filter is the double-SHA256 of the serialized filter. Filter headers are 32-byte hashes derived for each block filter. They are computed as the double-SHA256 of the concatenation of the filter hash with the previous filter header. The previous filter header used to calculate that of the genesis block is defined to be the 32-byte array of 0's.

New Messages

getcfilters `getcfilters` is used to request the compact filters of a particular type for a particular range of blocks. The message contains the following fields:

Field Name	Data Type	Byte Size	Description
FilterType	byte	1	Filter type for which headers are requested
StartHeight	uint32	4	The height of the first block in the requested range
StopHash	[32]byte	32	The hash of the last block in the requested range

[7]https://bitcoin.org/en/developer-guide#headers-first

Field Name	Data Type	Byte Size	Description

1. Nodes SHOULD NOT send `getcfilters` unless the peer has signaled support for this filter type. Nodes receiving `getcfilters` with an unsupported filter type SHOULD NOT respond.
2. StopHash MUST be known to belong to a block accepted by the receiving peer. This is the case if the peer had previously sent a `headers` or `inv` message with that block or any descendents. A node that receives `getcfilters` with an unknown StopHash SHOULD NOT respond.
3. The height of the block with hash StopHash MUST be greater than or equal to StartHeight, and the difference MUST be strictly less than 1000.
4. The receiving node MUST respond to valid requests by sending one `cfilter` message for each block in the requested range, sequentially in order by block height.

cfilter `cfilter` is sent in response to `getcfilters`, one for each block in the requested range. The message contains the following fields:

Field Name	Data Type	Byte Size	Description
FilterType	byte	1	Byte identifying the type of filter being returned
BlockHash	[32]byte	32	Block hash of the Bitcoin block for which the filter is be
NumFilterBytes	CompactSize	1-5	A variable length integer representing the size of the filte
FilterBytes	[]byte	NumFilterBytes	The serialized compact filter for this block

1. The FilterType SHOULD match the field in the `getcfilters` request, and BlockHash must correspond to a block that is an ancestor of StopHash with height greater than or equal to StartHeight.

getcfheaders `getcfheaders` is used to request verifiable filter headers for a range of blocks. The message contains the following fields:

Field Name	Data Type	Byte Size	Description
FilterType	byte	1	Filter type for which headers are requested
StartHeight	uint32	4	The height of the first block in the requested range
StopHash	[32]byte	32	The hash of the last block in the requested range

1. Nodes SHOULD NOT send `getcfheaders` unless the peer has signaled support for this filter type. Nodes receiving `getcfheaders` with an unsupported filter type SHOULD NOT respond.
2. StopHash MUST be known to belong to a block accepted by the receiving peer. This is the case if the peer had previously sent a `headers` or `inv` message with that block or any descendents. A node that receives `getcfheaders` with an unknown StopHash SHOULD NOT respond.

3. The height of the block with hash StopHash MUST be greater than or equal to StartHeight, and the difference MUST be strictly less than 2,000.

cfheaders `cfheaders` is sent in response to `getcfheaders`. Instead of including the filter headers themselves, the response includes one filter header and a sequence of filter hashes, from which the headers can be derived. This has the benefit that the client can verify the binding links between the headers. The message contains the following fields:

Field Name	Data Type	Byte Size	Description
FilterType	byte	1	Filter type for which hashes are requested
StopHash	[32]byte	32	The hash of the last block in the requested
PreviousFilterHeader	[32]byte	32	The filter header preceding the first block i
FilterHashesLength	CompactSize	1-3	The length of the following vector of filter
FilterHashes	[][32]byte	FilterHashesLength * 32	The filter hashes for each block in the requ

1. The FilterType and StopHash SHOULD match the fields in the `getcfheaders` request.
2. FilterHashesLength MUST NOT be greater than 2,000.
3. FilterHashes MUST have one entry for each block on the chain terminating with tip StopHash, starting with the block at height StartHeight. The entries MUST be the filter hashes of the given type for each block in that range, in ascending order by height.
4. PreviousFilterHeader MUST be set to the previous filter header of first block in the requested range.

getcfcheckpt `getcfcheckpt` is used to request filter headers at evenly spaced intervals over a range of blocks. Clients may use filter hashes from `getcfheaders` to connect these checkpoints, as is described in the Client Operation section below. The `getcfcheckpt` message contains the following fields:

Field Name	Data Type	Byte Size	Description
FilterType	byte	1	Filter type for which headers are requested
StopHash	[32]byte	32	The hash of the last block in the chain that headers are requested for

1. Nodes SHOULD NOT send `getcfcheckpt` unless the peer has signaled support for this filter type. Nodes receiving `getcfcheckpt` with an unsupported filter type SHOULD NOT respond.
2. StopHash MUST be known to belong to a block accepted by the receiving peer. This is the case if the peer had previously sent a `headers` or `inv` message with any descendent blocks. A node that receives `getcfcheckpt` with an unknown StopHash SHOULD NOT respond.

cfcheckpt `cfcheckpt` is sent in response to `getcfcheckpt`. The filter headers included are the set of all filter headers on the requested chain where the height is a positive multiple of 1,000. The message contains the following fields:

Field Name	Data Type	Byte Size	Description
FilterType	byte	1	Filter type for which headers are requeste
StopHash	[32]byte	32	The hash of the last block in the chain tha
FilterHeadersLength	CompactSize	1-3	The length of the following vector of filter
FilterHeaders	[][32]byte	FilterHeadersLength * 32	The filter headers at intervals of 1,000

1. The FilterType and StopHash SHOULD match the fields in the `getcfcheckpt` request.
2. FilterHeaders MUST have exactly one entry for each block on the chain terminating in StopHash, where the block height is a multiple of 1,000 greater than 0. The entries MUST be the filter headers of the given type for each such block, in ascending order by height.

Node Operation

Full nodes MAY opt to support this BIP and generate filters for any of the specified filter types. Such nodes SHOULD treat the filters as an additional index of the blockchain. For each new block that is connected to the main chain, nodes SHOULD generate filters for all supported types and persist them. Nodes that are missing filters and are already synced with the blockchain SHOULD reindex the chain upon start-up, constructing filters for each block from genesis to the current tip. They also SHOULD keep every checkpoint header in memory, so that `getcfcheckpt` requests do not result in many random-access disk reads.

Nodes SHOULD NOT generate filters dynamically on request, as malicious peers may be able to perform DoS attacks by requesting small filters derived from large blocks. This would require an asymmetical amount of I/O on the node to compute and serve, similar to attacks against BIP 37 enabled nodes noted in BIP 111.

Nodes MAY prune block data after generating and storing all filters for a block.

Client Operation

This section provides recommendations for light clients to download filters with maximal security.

Clients SHOULD first sync the entire block header chain from peers using the standard headers-first syncing mechanism before downloading any block filters or filter headers. Clients configured with trusted checkpoints MAY only sync headers started from the last checkpoint. Clients SHOULD disconnect any outbound peers whose best chain has significantly less work than the known longest proof-of-work chain.

Once a client's block headers are in sync, it SHOULD download and verify filter headers for all blocks and filter types that it might later download. The client SHOULD send `getcfheaders` messages to peers and derive and store the filter headers for each block. The client MAY first fetch headers at evenly spaced intervals of 1,000 by sending `getcfcheckpt`. The header checkpoints allow the client to download filter headers for different intervals from multiple peers in parallel, verifying each range of 1,000 headers against the checkpoints.

Unless securely connected to a trusted peer that is serving filter headers, the client SHOULD connect to multiple outbound peers that support each filter type to mitigate the risk of downloading incorrect headers. If the client receives conflicting filter headers from different peers for any block and filter type, it SHOULD interrogate them to determine which is faulty. The client SHOULD use `getcfheaders` and/or `getcfcheckpt` to first identify the first filter headers that the peers disagree on. The client then SHOULD download the full block from any peer and derive the correct filter and filter header. The client SHOULD ban any peers that sent a filter header that does not match the computed one.

Once the client has downloaded and verified all filter headers needed, *and* no outbound peers have sent conflicting headers, the client can download the actual block filters it needs. The client MAY backfill filter headers before the first verified one at this point if it only downloaded them starting at a later point. Clients SHOULD persist the verified filter headers for last 100 blocks in the chain (or whatever finality depth is desired), to compare against headers received from new peers after restart. They MAY store more filter headers to avoid redownloading them if a rescan is later necessary.

Starting from the first block in the desired range, the client now MAY download the filters. The client SHOULD test that each filter links to its corresponding filter header and ban peers that send incorrect filters. The client MAY download multiple filters at once to increase throughput, though it SHOULD test the filters sequentially. The client MAY check if a filter is empty before requesting it by checking if the filter header commits to the hash of the empty filter, saving a round trip if that is the case.

Each time a new valid block header is received, the client SHOULD request the corresponding filter headers from all eligible peers. If two peers send conflicting filter headers, the client should interrogate them as described above and ban any peers that send an invalid header.

If a client is fetching full blocks from the P2P network, they SHOULD be downloaded from outbound peers at random to mitigate privacy loss due to transaction intersection analysis. Note that blocks may be downloaded from peers that do not support this BIP.

Rationale

The filter headers and checkpoints messages are defined to help clients identify the correct filter for a block when connected to peers sending conflicting information. An alternative solution is to require Bitcoin blocks to include commitments to derived block filters, so light clients can verify authenticity given block headers and some additional witness data. This would require a network-wide change to the Bitcoin consensus rules, however, whereas this document proposes a solution purely at the P2P layer.

The constant interval of 1,000 blocks between checkpoints was chosen so that, given the current chain height and rate of growth, the size of a `cfcheckpt` message is not drastically different from a `cfheaders` message between two checkpoints. Also, 1,000 is a nice round number, at least to those of us who think in decimal.

Compatibility

This light client mode is not compatible with current node deployments and requires support for the new P2P messages. The node implementation of this proposal is not incompatible with the current P2P network rules (ie. doesn't affect network topology of full nodes). Light clients may adopt protocols based on this as an alternative to the existing BIP 37. Adoption of this BIP may result in reduced network support for BIP 37.

Acknowledgments

We would like to thank bfd (from the bitcoin-dev mailing list) for bringing the basis of this BIP to our attention, Joseph Poon for suggesting the filter header chain scheme, and Pedro Martelletto for writing the initial indexing code for `btcd`.

We would also like to thank Dave Collins, JJ Jeffrey, Eric Lombrozo, and Matt Corallo for useful discussions.

Reference Implementation

Light client: 1

Full-node indexing: https://github.com/Roasbeef/btcd/tree/segwit-cbf

Golomb-Rice Coded sets: https://github.com/Roasbeef/btcutil/tree/gcs/gcs

References

Copyright