```
BIP: 158
Layer: Peer Services
Title: Compact Block Filters for Light Clients
Author: Olaoluwa Osuntokun <laolu32@gmail.com>
        Alex Akselrod <alex@akselrod.org>
Comments-Summary: None yet
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0158
Status: Draft
Type: Standards Track
Created: 2017-05-24
License: CC0-1.0
```

Abstract

This BIP describes a structure for compact filters on block data, for use in the BIP 157 light client protocol[1]. The filter construction proposed is an alternative to Bloom filters, as used in BIP 37, that minimizes filter size by using Golomb-Rice coding for compression. This document specifies one initial filter type based on this construction that enables basic wallets and applications with more advanced smart contracts.

Motivation

BIP 157 defines a light client protocol based on deterministic filters of block content. The filters are designed to minimize the expected bandwidth consumed by light clients, downloading filters and full blocks. This document defines the initial filter type *basic* that is designed to reduce the filter size for regular wallets.

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.

Data pushes are byte vectors pushed to the stack according to the rules of Bitcoin script.

Bit streams are readable and writable streams of individual bits. The following functions are used in the pseudocode in this document:

- `new_bit_stream` instantiates a new writable bit stream
- `new_bit_stream(vector)` instantiates a new bit stream reading data from `vector`
- `write_bit(stream, b)` appends the bit `b` to the end of the stream

[1] bip-0157.mediawiki

- `read_bit(stream)` reads the next available bit from the stream
- `write_bits_big_endian(stream, n, k)` appends the `k` least significant bits of integer `n` to the end of the stream in big-endian bit order
- `read_bits_big_endian(stream, k)` reads the next available `k` bits from the stream and interprets them as the least significant bits of a big-endian integer

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

Golomb-Coded Sets

For each block, compact filters are derived containing sets of items associated with the block (eg. addresses sent to, outpoints spent, etc.). A set of such data objects is compressed into a probabilistic structure called a *Golomb-coded set* (GCS), which matches all items in the set with probability 1, and matches other items with probability `1/M` for some integer parameter `M`. The encoding is also parameterized by `P`, the bit length of the remainder code. Each filter defined specifies values for `P` and `M`.

At a high level, a GCS is constructed from a set of `N` items by:

1. hashing all items to 64-bit integers in the range `[0, N * M)`
2. sorting the hashed values in ascending order
3. computing the differences between each value and the previous one
4. writing the differences sequentially, compressed with Golomb-Rice coding

The following sections describe each step in greater detail.

Hashing Data Objects The first step in the filter construction is hashing the variable-sized raw items in the set to the range `[0, F)`, where `F = N * M`. Customarily, `M` is set to `2^P`. However, if one is able to select both Parameters independently, then more optimal values can be selected[2]. Set membership queries against the hash outputs will have a false positive rate of `M`. To avoid integer overflow, the number of items `N` MUST be $<2^32$ and `M` MUST be $<2^32$.

The items are first passed through the pseudorandom function *SipHash*, which takes a 128-bit key `k` and a variable-sized byte vector and produces a uniformly random 64-bit output. Implementations of this BIP MUST use the SipHash parameters `c = 2` and `d = 4`.

The 64-bit SipHash outputs are then mapped uniformly over the desired range by multiplying with F and taking the top 64 bits of the 128-bit result. This algorithm is a faster alternative to modulo reduction, as it avoids the expensive division

[2] https://gist.github.com/sipa/576d5f09c3b86c3b1b75598d799fc845

operation[3]. Note that care must be taken when implementing this reduction to ensure the upper 64 bits of the integer multiplication are not truncated; certain architectures and high level languages may require code that decomposes the 64-bit multiplication into four 32-bit multiplications and recombines into the result.

```
hash_to_range(item: []byte, F: uint64, k: [16]byte) -> uint64:
    return (siphash(k, item) * F) >> 64

hashed_set_construct(raw_items: [][]byte, k: [16]byte, M: uint) -> []uint64:
    let N = len(raw_items)
    let F = N * M

    let set_items = []

    for item in raw_items:
        let set_value = hash_to_range(item, F, k)
        set_items.append(set_value)

    return set_items
```

Golomb-Rice Coding Instead of writing the items in the hashed set directly to the filter, greater compression is achieved by only writing the differences between successive items in sorted order. Since the items are distributed uniformly, it can be shown that the differences resemble a geometric distribution[4]. *Golomb-Rice coding*[5] is a technique that optimally compresses geometrically distributed values.

With Golomb-Rice, a value is split into a quotient and remainder modulo `2^P`, which are encoded separately. The quotient `q` is encoded as *unary*, with a string of `q` 1's followed by one 0. The remainder `r` is represented in big-endian by P bits. For example, this is a table of Golomb-Rice coded values using `P=2`:

n	(q, r)	c
0	(0, 0)	`0 00`
1	(0, 1)	`0 01`
2	(0, 2)	`0 10`
3	(0, 3)	`0 11`
4	(1, 0)	`10 00`
5	(1, 1)	`10 01`
6	(1, 2)	`10 10`
7	(1, 3)	`10 11`
8	(2, 0)	`110 00`
9	(2, 1)	`110 01`

[3] https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/
[4] https://en.wikipedia.org/wiki/Geometric_distribution
[5] https://en.wikipedia.org/wiki/Golomb_coding#Rice_coding

n	(q, r)	c

```
golomb_encode(stream, x: uint64, P: uint):
    let q = x >> P

    while q > 0:
        write_bit(stream, 1)
        q--
    write_bit(stream, 0)

    write_bits_big_endian(stream, x, P)

golomb_decode(stream, P: uint) -> uint64:
    let q = 0
    while read_bit(stream) == 1:
        q++

    let r = read_bits_big_endian(stream, P)

    let x = (q << P) + r
    return x
```

Set Construction A GCS is constructed from four parameters:

- `L`, a vector of `N` raw items
- `P`, the bit parameter of the Golomb-Rice coding
- `M`, the target false positive rate
- `k`, the 128-bit key used to randomize the SipHash outputs

The result is a byte vector with a minimum size of `N * (P + 1)` bits.

The raw items in `L` are first hashed to 64-bit unsigned integers as specified above and sorted. The differences between consecutive values, hereafter referred to as *deltas*, are encoded sequentially to a bit stream with Golomb-Rice coding. Finally, the bit stream is padded with 0's to the nearest byte boundary and serialized to the output byte vector.

```
construct_gcs(L: [][]byte, P: uint, k: [16]byte, M: uint) -> []byte:
    let set_items = hashed_set_construct(L, k, M)

    set_items.sort()

    let output_stream = new_bit_stream()

    let last_value = 0
    for item in set_items:
```

```
        let delta = item - last_value
        golomb_encode(output_stream, delta, P)
        last_value = item

    return output_stream.bytes()
```

Set Querying/Decompression To check membership of an item in a compressed GCS, one must reconstruct the hashed set members from the encoded deltas. The procedure to do so is the reverse of the compression: deltas are decoded one by one and added to a cumulative sum. Each intermediate sum represents a hashed value in the original set. The queried item is hashed in the same way as the set members and compared against the reconstructed values. Note that querying does not require the entire decompressed set be held in memory at once.

```
gcs_match(key: [16]byte, compressed_set: []byte, target: []byte, P: uint, N: uint, M: uint)
    let F = N * M
    let target_hash = hash_to_range(target, F, k)

    stream = new_bit_stream(compressed_set)

    let last_value = 0

    loop N times:
        let delta = golomb_decode(stream, P)
        let set_item = last_value + delta

        if set_item == target_hash:
            return true

        // Since the values in the set are sorted, terminate the search once
        // the decoded value exceeds the target.
        if set_item > target_hash:
            break

        last_value = set_item

    return false
```

Some applications may need to check for set intersection instead of membership of a single item. This can be performed far more efficiently than checking each item individually by leveraging the sorted structure of the compressed GCS. First the query elements are all hashed and sorted, then compared in order against the decompressed GCS contents. See Appendix B for pseudocode.

Block Filters

This BIP defines one initial filter type:

- Basic (`0x00`)
  - `M = 784931`
  - `P = 19`

Contents The basic filter is designed to contain everything that a light client needs to sync a regular Bitcoin wallet. A basic filter MUST contain exactly the following items for each transaction in a block:

- The previous output script (the script being spent) for each input, except for the coinbase transaction.
- The scriptPubKey of each output, aside from all `OP_RETURN` output scripts.

Any "nil" items MUST NOT be included into the final set of filter elements.

We exclude all outputs that start with `OP_RETURN` in order to allow filters to easily be committed to in the future via a soft-fork. A likely area for future commitments is an additional `OP_RETURN` output in the coinbase transaction similar to the current witness commitment [6]. By excluding all `OP_RETURN` outputs we avoid a circular dependency between the commitment, and the item being committed to.

Construction The basic type is constructed as Golomb-coded sets with the following parameters.

The parameter `P` MUST be set to `19`, and the parameter `M` MUST be set to `784931`. Analysis has shown that if one is able to select `P` and `M` independently, then setting `M=1.497137 * 2^P` is close to optimal [7].

Empirical analysis also shows that these parameters minimize the bandwidth utilized, considering both the expected number of blocks downloaded due to false positives and the size of the filters themselves.

The parameter `k` MUST be set to the first 16 bytes of the hash (in standard little-endian representation) of the block for which the filter is constructed. This ensures the key is deterministic while still varying from block to block.

Since the value `N` is required to decode a GCS, a serialized GCS includes it as a prefix, written as a `CompactSize`. Thus, the complete serialization of a filter is:

- `N`, encoded as a `CompactSize`
- The bytes of the compressed filter itself

[6] https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki
[7] https://gist.github.com/sipa/576d5f09c3b86c3b1b75598d799fc845

Signaling This BIP allocates a new service bit:

NODE_COMPACT_FILTERS	`1 << 6`	If enabled, the node MUST respond to all BIP 157 messages for f

Compatibility

This block filter construction is not incompatible with existing software, though it requires implementation of the new filters.

Acknowledgments

We would like to thank bfd (from the bitcoin-dev mailing list) for bringing the basis of this BIP to our attention, Greg Maxwell for pointing us in the direction of Golomb-Rice coding and fast range optimization, Pieter Wullie for his analysis of optimal GCS parameters, and Pedro Martelletto for writing the initial indexing code for `btcd`.

We would also like to thank Dave Collins, JJ Jeffrey, and Eric Lombrozo 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/btcsuite/btcutil/blob/master/gcs

Appendix A: Alternatives

A number of alternative set encodings were considered before Golomb-coded sets were settled upon. In this appendix section, we'll list a few of the alternatives along with our rationale for not pursuing them.

Bloom Filters Bloom Filters are perhaps the best known probabilistic data structure for testing set membership, and were introduced into the Bitcoin protocol with BIP 37. The size of a Bloom filter is larger than the expected size of a GCS with the same false positive rate, which is the main reason the option was rejected.

Cryptographic Accumulators Cryptographic accumulators[8] are a cryptographic data structures that enable (amongst other operations) a one way membership test. One advantage of accumulators are that they are constant size, independent of the number of elements inserted into the accumulator. However, current constructions of cryptographic accumulators require an initial trusted set

[8] https://en.wikipedia.org/wiki/Accumulator_(cryptography)

up. Additionally, accumulators based on the Strong-RSA Assumption require mapping set items to prime representatives in the associated group which can be preemptively expensive.

Matrix Based Probabilistic Set Data Structures There exist data structures based on matrix solving which are even more space efficient compared to Bloom filters[9]. We instead opted for our GCS-based filters as they have a much lower implementation complexity and are easier to understand.

Appendix B: Pseudocode

Golomb-Coded Set Multi-Match

```
gcs_match_any(key: [16]byte, compressed_set: []byte, targets: [][]byte, P: uint, N: uint, M:
    let F = N * M

    // Map targets to the same range as the set hashes.
    let target_hashes = []
    for target in targets:
        let target_hash = hash_to_range(target, F, k)
        target_hashes.append(target_hash)

    // Sort targets so matching can be checked in linear time.
    target_hashes.sort()

    stream = new_bit_stream(compressed_set)

    let value = 0
    let target_idx = 0
    let target_val = target_hashes[target_idx]

    loop N times:
        let delta = golomb_decode(stream, P)
        value += delta

        inner loop:
            if target_val == value:
                return true

            // Move on to the next set value.
            else if target_val > value:
                break inner loop

            // Move on to the next target value.
            else if target_val < value:
```

[9] https://arxiv.org/pdf/0804.1845.pdf

```
                target_idx++

                // If there are no targets left, then there are no matches.
                if target_idx == len(targets):
                    break outer loop

                target_val = target_hashes[target_idx]

    return false
```

Appendix C: Test Vectors

Test vectors for basic block filters on five testnet blocks, including the filters and filter headers, can be found here. The code to generate them can be found here.

References

Copyright