hippiepug Documentation

Release 0.5.0

Bogdan Kulynych

Oct 22, 2019

Contents

Sublinear-lookup blockchains and efficient key-value Merkle trees

This library provides implementations of two cryptographic data structures:

- Blockchains with log(n) sublinear traversal, implemented as high-integrity deterministic skip-lists (skipchains). In this kind of blockchain verifying that block *b* extends block *a* does not require to download and process all blocks between *a* and *b*, but only a logarithmic amount of them.
- Verifiable dictionary, implemented as a key-value Merkle tree that guarantees unique resolution. A proof of inclusion of a key-value pair in such a tree also proves that there does not exist another value for a given key somewhere else in the tree.

Both are meant to be used with a content-addressable storage. Each data structure supports logarithmic queries, and logarithmic proofs of inclusion:

	Retrievals per lookup	Inclusion proof size	Append
Skipchain	O(log(n))	O(log(n))	O(1)
Key-value Merkle tree	O(log(n))	O(log(n))	Immutable

with *n* being the size of the dictionary, or the number of blocks in the case of a chain.

The theoretical details are in the paper.

CHAPTER 1

Getting started

You can install the library from PyPI:

```
pip install hippiepug
```

Then, the easiest way to run the tests is:

```
python setup.py test
```

Be sure to check out the usage guide.

CHAPTER 2

Usage guide

2.1 Overview

Skipchain. *hippiepug.chain* implements a blockchain that only requires logarithmic-size proofs of inclusion. Each block of such blockchain has not one but many hash-pointers to previous blocks.

Key-value Merkle tree. *hippiepug.tree* implements a verifiable dictionary as a key-value Merkle tree that guarantees unique resolution. For each lookup key, one can produce a proof of inclusion of the key in the tree. Moreover, unique resolution ensures that a proof of inclusion also proves that no other value with the same lookup key exists somewhere else in the tree.

Object store. *hippiepug.store* implements a content-addressable key-value store in which keys are cryptographic hashes of values. Using such storage with `hippiepug` data structures is convenient, because a creator of a chain or a tree only needs to provide a querier with a hash of a chain head or a tree root. That is, there is no need to explicitly produce and transmit inclusion proofs. Queriers will be able to verify inclusion on the fly, provided the storage is available. See section "*Producing and verifying proofs*" for more.

2.2 Using object stores

`hippiepug` includes an instantation of an in-memory content-addressable storage that uses SHA256 for hashes: *hippiepug.store.Sha256DictStore*. By default, the hashes are truncated to 8 bytes.

```
from hippiepug.store import Sha256DictStore

store = Sha256DictStore()
obj = b'dummy'
obj_hash = store.hash_object(obj)  # 'b5a2c96250612366'

store.add(obj) == obj_hash  # True
```

The store verifies hashes internally on each lookup.

```
obj_hash in store  # True
store.get(obj_hash) == obj   # True
```

If you want to use external storage, you can provide a dict-like facade to it and pass as a `backend` parameter:

```
class CustomBackend(object):

    def get(self, k):
        return 'stub'

    def __setitem__(self, k, v):
        pass

store = Sha256DictStore(backend=CustomBackend())
```

To change the hash function, subclass *hippiepug.store.BaseDictStore*, and implement the `hash_object` method.

You can also define a completely different store by implementing abstract base *hippiepug.store.BaseStore*.

2.3 Building the data structures

2.3.1 Chain

To append a new block to a chain, first obtain an existing chain, or initialize a new empty *hippiepug.chain.Chain* object:

```
from hippiepug.chain import Chain

chain = Chain(store)
chain.head  # None
```

Then, add chain blocks ony by one.

```
from hippiepug.chain import BlockBuilder

block_builder = BlockBuilder(chain)
block_builder.payload = b'This is the first block!'
block_builder.commit()

chain.head  # '154bdee593d8c9b2'
```

You can continue adding blocks using the same builder instance.

```
block_builder.payload  # None
block_builder.payload = b'This is the second block!'
block_builder.commit()

chain.head  # '48e399de59796ab1'
```

The builder automatically fills all the skipchain special block attributes, like hashes of previous blocks.

2.3.2 Tree

Unlike chains, `hippiepug` trees can not be extended. To build a new tree, initialize the tree builder on a store, and set the key-value pairs to be committed.

```
from hippiepug.tree import TreeBuilder

tree_builder = TreeBuilder(store)
tree_builder['foo'] = b'bar'
tree_builder['baz'] = b'wow'
```

Once all key-value pairs are added, commit them to store and obtain a view of the committed tree:

```
tree = tree_builder.commit()
tree.root  # '150cc8da6d6cfa17'
```

2.4 Querying the data structures

2.4.1 Chain

To get a queryable view of a chain, you need to specify the storage where its blocks reside, and the head of the chain (hash of the latest block). You can then retrieve blocks by their indices, or iterate.

```
chain = Chain(store, head='48e399de59796ab1')
first_block = chain[0]
first_block.payload  # b'This is the first block!'

for block in chain:
    print(block.index)  # will print 1, and then 0
```

You can also get the latest view of a current chain while building a block in `block_builder.chain`.

2.4.2 Tree

Similarly, to get a view of a tree, you need to specify the storage, and the root of the tree (hash of the root node). You can then retrieve stored values by corresponding lookup keys.

```
from hippepug.tree import Tree

tree = Tree(store, root='150cc8da6d6cfa17')
tree['foo']  # b'bar'
'baz' in tree  # True
```

2.5 Producing and verifying proofs

When the creator of a data structure and the querier use the same storage (e.g., external database), no additional work regarding inclusion proofs needs to be done, since queries produce inclusion proofs on the fly. This scenario, however, is not always possible. In such case, `hippiepug` allows to produce and verify proofs explicitly.

2.5.1 Chain

You can get the proof of block inclusion from a chain view:

```
block, proof = chain.get_block_by_index(0, return_proof=True)
```

A proof is a subset of blocks between head block and the requested block.

To verify the proof, the querier needs to locally reproduce a store, populating it with the blocks in the proof, and then query the chain in the reproduced store normally. A convenience utility *hippiepug.chain.verify_chain_inclusion_proof()* does all of this internally, and only returns the verification result:

```
from hippiepug.chain import verify_chain_inclusion_proof

verification_store = Sha256DictStore()
verify_chain_inclusion_proof(verification_store,
                             chain.head, block, proof)  # True.
```

2.5.2 Tree

You can get the proof of value and lookup key inclusion from a tree view:

```
value, proof = tree.get_value_by_lookup_key('foo', return_proof=True)
```

For trees, a proof is the list of nodes on the path from root to the leaf containing the lookup key.

The mechanism of verifying an explicit proof is the same as with chains: locally reproduce a store populating it with all the nodes in the proof, and then query normally the tree in the reproduced store. Similarly, a utility *hippiepug.tree.verify_tree_inclusion_proof()* does this internally and returns the verification result:

```
from hippiepug.tree import verify_tree_inclusion_proof

verification_store = Sha256DictStore()
verify_tree_inclusion_proof(verification_store, tree.root,
                            lookup_key='foo', value=b'bar',
                            proof=proof)  # True.
```

2.6 Serialization

`hippiepug` includes default binary serialization using `msgpack` library.

```
from hippiepug.pack import decode, encode

block = chain[0]
decode(encode(block)) == block  # True
```

If you want to define custom serializers, be sure to check the documentation of *hippiepug.pack*. You need to be careful with custom encoders to not jeopardize security of the data structure.

Once you have defined a custom encoder and decoder, you can set them to global defaults like this:

```
from hippiepug.pack import EncodingParams

my_params = EncodingParams()
```

(continues on next page)

(continued from previous page)

```
my_params.encoder = lambda obj: b'encoded!'
my_params.decoder = lambda encoded: b'decoded!'

EncodingParams.set_global_default(my_params)
```

Alternatively, you can also limit their usage to a specific context:

```
with my_params.as_default():
    encode(b'stub')  # b'encoded!'
```

CHAPTER 3

API

3.1 Chain

Tools for building and interpreting skipchains.

class hippiepug.chain.**BlockBuilder**(*chain*)

Customizable builder of skipchain blocks.

You can override the pre-commit hook (*BlockBuilder.pre_commit()*) to modify the payload before the block is committed to a chain. This is needed, say, if you want to sign the payload before commiting.

Parameters **chain** – Chain to which the block should belong.

Set the payload before committing:

```
>>> from .store import Sha256DictStore
>>> store = Sha256DictStore()
>>> chain = Chain(store)
>>> builder = BlockBuilder(chain)
>>> builder.payload = b'Hello, world!'
>>> block = builder.commit()
>>> block == chain.head_block
True
```

chain

The associated chain.

commit()

Commit the block to the associated chain.

Returns The block that was committed.

fingers

Anticipated skip-list fingers (back-pointers to previous blocks).

index

Anticipated index of the block being built.

payload
Anticipated block payload.

pre_commit()
Pre-commit hook.

This can be overriden. For example, you can add a signature that includes index and fingers into your payload before the block is committed.

static skipchain_indices(*index*)
Finger indices for a given index.

Parameters **index** (*int>=0*) – Block index

class hippiepug.chain.**Chain**(*object_store*, *head=None*, *cache=None*)
Skipchain (hash chain with skip-list pointers).

To add a new block to a chain, use *BlockBuilder*.

Warning: All read accesses are cached. The cache is assumed to be trusted, so blocks retrieved from cache are not checked for integrity, unlike when they are retrieved from the object store.

See also:

- *hippiepug.tree.Tree*

class ChainIterator(*current_index*, *chain*)
Chain iterator.

Note: Iterates in the reverse order: latest block first.

__getitem__(*index*)
Get block by index.

get_block_by_index(*index*, *return_proof=False*)
Get block by index.

Optionally returns inclusion proof, that is a list of intermediate blocks, sufficient to verify the inclusion of the retrieved block.

Parameters

- **index** (*int>=0*) – Block index
- **return_proof** (*bool*) – Whether to return inclusion proof

Returns Found block or None, or (block, proof) tuple if return_proof is True.

Raises If the index is out of bounds, raises IndexError.

head_block
The latest block in the chain.

hippiepug.chain.**verify_chain_inclusion_proof**(*store*, *head*, *block*, *proof*)
Verify inclusion proof for a block on a chain.

Parameters

- **store** – Object store, may be empty
- **head** – Chain head

- **block** – Block
- **proof** (*list of decoded blocks*) – Inclusion proof

Returns bool

3.2 Tree

Tools for building and interpreting key-value Merkle trees.

class hippiepug.tree.**Tree**(*object_store*, *root*, *cache=None*)
View of a Merkle tree.

Use *TreeBuilder* to build a Merkle tree first.

Parameters

- **object_store** – Object store
- **root** – The hash of the root node
- **cache** (*dict*) – Cache

Warning: All read accesses are cached. The cache is assumed to be trusted, so blocks retrieved from cache are not checked for integrity, unlike when they are retrieved from the object store.

See also:

- *hippiepug.chain.Chain*

__contains__(*lookup_key*)
Check if lookup key is in the tree.

__getitem__(*lookup_key*)
Retrieve value by its lookup key.

Returns Corresponding value

Raises KeyError when the lookup key was not found.

get_value_by_lookup_key(*lookup_key*, *return_proof=False*)
Retrieve value by its lookup key.

Parameters

- **lookup_key** – Lookup key
- **return_proof** – Whether to return inclusion proof

Returns Only the value when return_proof is False, and a (value, proof) tuple when return_proof is True. A value is None when the lookup key was not found.

root_node
The root node.

class hippiepug.tree.**TreeBuilder**(*object_store*)
Builder for a key-value Merkle tree.

Parameters **object_store** – Object store

You can add items using a dict-like interface:

```
>>> from .store import Sha256DictStore
>>> store = Sha256DictStore()
>>> builder = TreeBuilder(store)
>>> builder['foo'] = b'bar'
>>> builder['baz'] = b'zez'
>>> tree = builder.commit()
>>> 'foo' in tree
True
```

__setitem__(*lookup_key*, *value*)
Add item for committing to the tree.

commit()
Commit items to the tree.

hippiepug.tree.**verify_tree_inclusion_proof**(*store*, *root*, *lookup_key*, *value*, *proof*)
Verify inclusion proof for a tree.

Parameters

- **store** – Object store, may be empty
- **head** – Tree root
- **lookup_key** – Lookup key
- **value** – Value associated with the lookup key
- **proof** (*tuple containing list of decoded path nodes*) – Inclusion proof

Returns bool

3.3 Store

class hippiepug.store.**BaseDictStore**(*backend=None*)
Store with dict-like backend.

Parameters **backend** (*dict-like*) – Backend

__contains__(*obj_hash*)
Check if obj with a given hash is in the store.

add(*serialized_obj*)
Add an object to the store.

If an object with this hash already exists, silently does nothing.

get(*obj_hash*, *check_integrity=True*)
Get an object with a given hash from the store.

If the object does not exist, returns None.

Parameters

- **obj_hash** – ASCII hash of the object
- **check_integrity** – Whether to check the hash of the retrieved object against the given hash.

class hippiepug.store.**BaseStore**
Abstract base class for a content-addresable store.

__contains__(*obj_hash*)
Check whether the store contains an object with a give hash.

Parameters **obj_hash** – ASCII hash

add(*serialized_obj*)
Put the object in the store.

Parameters **serialized_obj** – Object, serialized to bytes

Returns Hash of the object.

get(*obj_hash*, *check_integrity=True*)
Return the object by its ASCII hash value.

Parameters

- **obj_hash** – ASCII hash
- **check_integrity** – Whether to check the hash upon retrieval

classmethod hash_object(*serialized_obj*)
Return the ASCII hash of the object.

Parameters **obj** – Object, serialized to bytes

exception hippiepug.store.**IntegrityValidationError**

class hippiepug.store.**Sha256DictStore**(*backend=None*)
Dict-based store using truncated SHA256 hex-encoded hashes.

```
>>> store = Sha256DictStore()
>>> obj = b'dummy'
>>> obj_hash = store.hash_object(obj)
>>> store.add(obj) == obj_hash
True
>>> obj_hash in store
True
>>> b'nonexistent' not in store
True
>>> store.get(obj_hash) == obj
True
```

hash_object(*serialized_obj*)
Return a SHA256 hex-encoded hash of a serialized object.

3.4 Basic containers

Basic building blocks.

class hippiepug.struct.**ChainBlock**(*payload*, *index=0*, *fingers=NOTHING*)
Skipchain block.

Parameters

- **payload** – Block payload
- **index** – Block index
- **fingers** – Back-pointers to previous blocks

class hippiepug.struct.**TreeLeaf**(*lookup_key=None*, *payload_hash=None*)
Merkle tree leaf.

Parameters

- **lookup_key** – Lookup key
- **payload_hash** – Hash of the payload

class hippiepug.struct.**TreeNode**(*pivot_prefix*, *left_hash=None*, *right_hash=None*)
Merkle tree intermediate node.

Parameters

- **pivot_prefix** – Pivot key for the subtree
- **left_hash** – Hash of the left child
- **right_hash** – Hash of the right child

3.5 Serialization

Serializers for chain blocks and tree nodes.

Warning: You need to take extra care when defining custom serializations. Be sure that your serialization includes all the fields in the original structure. E.g., for chain blocks:

- `self.index`
- `self.fingers`
- Your payload

Unless this is done, the integrity of the data structures is screwed, since it's the serialized versions of nodes and blocks that are hashed.

class hippiepug.pack.**EncodingParams**(*encoder=NOTHING*, *decoder=NOTHING*)
Thread-local container for default encoder and decoder funcs.

Parameters

- **encoder** – Default encoder
- **decoder** – Default decoder

This is how you can override the defaults using this class:

```
>>> my_params = EncodingParams()
>>> my_params.encoder = lambda obj: b'encoded!'
>>> my_params.decoder = lambda encoded: b'decoded!'
>>> EncodingParams.set_global_default(my_params)
>>> encode(b'dummy') == b'encoded!'
True
>>> decode(b'encoded!') == b'decoded!'
True
>>> EncodingParams.reset_defaults()
```

hippiepug.pack.**decode**(*serialized*, *decoder=None*)
Deserialize object.

Parameters

- **serialized** – Encoded structure
- **encoder** – Custom de-serializer

hippiepug.pack.**encode**(*obj*, *encoder=None*)
Serialize object.

Parameters

- **obj** – Chain block, tree node, or bytes
- **encoder** – Custom serializer

hippiepug.pack.**msgpack_decoder**(*serialized_obj*)
Deserialize structure from msgpack-encoded tuple.

Default decoder.

hippiepug.pack.**msgpack_encoder**(*obj*)
Represent structure as tuple and serialize using msgpack.

Default encoder.

CHAPTER 4

Contributing

4.1 Dev setup

To install the development dependencies, clone the package from Github, and run within the folder:

```
pip install -e ".[dev]"
```

You can then run the tests from the root folder:

```
pytest
```

You can also run the tests against multiple pythons:

```
tox
```

Note that this invocation is expected to fail in the coverage upload stage (it needs access token to upload coverage report)

To build the documentation, run `make html` from the docs folder:

```
cd docs
make html
```

Then you can run a static HTML server from `docs/build/html`.

```
cd build/html
python -m http.server
```

CHAPTER 5

License

5.1 Notice

Some of the code was adapted from G.Danezis's hippiehug library: https://github.com/gdanezis/rousseau-chain

The license of hippiehug is reproduced below:

CHAPTER 6

Acknowledgements

- The library is a reimplementation of G. Danezis's hippiehug (hence the name).
- This work is funded by the NEXTLEAP project within the European Union's Horizon 2020 Framework Programme for Research and Innovation (H2020-ICT-2015, ICT-10-2015) under grant agreement 688722.
- The hippie pug logo kindly donated by M. Naiem.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

T

V