pymerkle

Release 6.1.0

fmerg

Aug 30, 2023

CONTENTS

CHAPTER

ONE

MERKLE-TREE IN PYTHON

Storage agnostic implementation capable of generating inclusion and consistency proofs.

1.1 Installation

```
pip install pymerkle
```

This will also install cachetools as a dependency.

1.2 Basic API

Let `MerkleTree` be any class implementing the `BaseMerkleTree` interface; e.g.,

```python
from pymerkle import InmemoryTree as MerkleTree

tree = MerkleTree(algorithm='sha256')
```

Append data into the tree and retrieve the corresponding hash value:

```python
index = tree.append_entry(b'foo')    # leaf index

value = tree.get_leaf(index)         # leaf hash
```

Current tree size:

```python
size = tree.get_size()    # number of leaves
```

Current and intermediate state:

```python
state = tree.get_state()    # current root-hash

state = tree.get_state(5)   # root-hash of size 5 subtree
```

1.2.1 Inclusion proof

Prove inclusion of the 3-rd leaf hash in the subtree of size 5:

```
proof = tree.prove_inclusion(3, 5)
```

Verify the proof against the base hash and the subtree root:

```
from pymerkle import verify_inclusion

base = tree.get_leaf(3)
root = tree.get_state(5)

verify_inclusion(base, root, proof)
```

1.2.2 Consistency proof

Prove consistency between the states with size 3 and 5:

```
proof = tree.prove_consistency(3, 5)
```

Verify the proof against the respective root hashes:

```
from pymerkle import verify_consistency

state1 = tree.get_state(3)
state2 = tree.get_state(5)

verify_consistency(state1, state2, proof)
```

1.3 Security

This library requires security review.

1.3.1 Resistance against second-preimage attack

This consists in the following standard technique:

- Upon computing the hash of a leaf node, prepend `0x00` to the payload
- Upon computing the hash of an interior node, prepend `0x01` to the payload

1.3.2 Resistance against CVE-2012-2459 DOS

Contrary to the bitcoin specification, lonely leaves are not duplicated while the tree is growing. Instead, a bifurcation node is created at the rightmost branch (see next section). As a consequence, the present implementation should be invulnerable to the CVE-2012-2459 DOS attack (see also here for insight).

1.4 Topology

Interior nodes are not assumed to be stored anywhere and no concrete links are created between them. The tree structure is determined by the recursive function which computes intermediate states on the fly and is essentially the same as RFC 9162 (Section 2). It turns out to be that of a binary Sakura tree (Section 5.4).

1.5 Storage

This library is unopinionated on how leaves are appended to the tree, i.e., how data is stored in concrete. Cryptographic functionality is encapsulated in the `BaseMerkleTree` abstract class, which admits pluggable storage backends through subclassing. It is the the developer's choice to decide how to store data by implementing the interior storage interface of this class. Any contiguously indexed dataset should do the job. Conversely, given any such dataset, we should be able to trivially implement a Merkle-tree that is operable with it.

1.6 Optimizations

The performance of a Merkle-tree depends on how efficiently it computes the root-hash for arbitrary leaf ranges. The recursive version of this function is slow (e.g., RFC 9162, Section 2).

This operation can be optimized using iterations on ranges whose size is a power of two. This has the effect of making proof generation five times faster, while peak memory usage remains reasonably low and sublinear with respect to size. Further boost is given by caching. Practically, a pretty big tree with sufficiently long uptime will respond instantly with negligible penalty in memory usage.

1.6.1 Public API

Initialization

Although pymerkle comes with concrete tree implementations, its primary purpose is to provide an abstract base class that encapsulates the cryptographic functionality of a Merkle-tree:

```
from pymerkle import BaseMerkleTree
```

Concrete implementations should inherit from this class and implement its internal abstract interface. This amounts to customizing leaf storage according to any desired application logic.

Superclass initialization

Initialization of `BaseMerkleTree` accepts the options shown below:

```
class MerkleTree(BaseMerkleTree):

    def __init__(self, *args, **kwargs)
        ...

        super().__init__(
            algorithm='sha256',
            disable_security=False,
            disable_optimizations=False,
            disable_cache=False,
            threshold=128,
            capacity=1024 ** 3
        )

    ...
```

- `algorithm`: specifies the hash function used by the tree. Defaults to *sha256*.
- `disable_security`: if *True*, resistance against second-preimage attack will be deactivated. Use it only for testing or debugging purposes. Defaults to *False*.
- `disable_optimizations`: if *True*, low-level computations will fallback to recursive unoptimized functions, similar to those described in RFC 9162. Use it for comparison purposes. Defaults to *False*.
- `disable_cache`: if *True*, the results of optimized low-level computations will not be cached. Use it for comparison purposes. Defaults to *False*.
- `theshold`: specifies which outputs of a low-level computation must be cached depending on the input of the computation. Refer *here* for the exact meaning of this parameter. Defaults to *128*.
- `capacity`: cache capacity in bytes. Defaults to 1GiB (which should be overabundant for any imaginable use case).

See *here* to see how to implement a Merkle-tree in detail.

Supported hash functions

`sha224`, `sha256`, `sha384`, `sha512`, `sha3_224`, `sha3_256`, `sha3_384`, `sha3_512`

Support for Keccak beyond SHA3

Installing pysha3 makes available following hash functions:

`keccak_224`, `keccak_256`, `keccak_384`, `keccak_512`

Warning: Requesting anything except for these raises a `ValueError`.

Concrete classes

Pymerkle provides two concrete implementations of `BaseMerkleTree` out of the box.

`InmemoryTree` is a non-persistent implementation where nodes are stored at runtime, intended for investigating and visualising the tree structure:

```
from pymerkle import InmemoryTree

tree = InmemoryTree(algorithm='sha256')
```

`SqliteTree` is a persistent implementation using a SQLite database as storage, intended for leightweight local applications:

```
from pymerkle import SqliteTree

tree = SqliteTree('merkle.db', algorithm='sha256')
```

This will open a connection to the specified database file (after creating it if not already existent). Alternatively, you can create an in-memory database as follows:

```
tree = SqliteTree(':memory:', algorithm='sha256')
```

Both trees are designed to accept data in binary format and hash it without further processing. See *here* for more details on these classes.

Entries

Entries are appended to the tree as leaves with contiguously increasing index. The exact type of entries depends on the particular implementation.

Note: In what follows, it is assumed without loss of generality that the tree accepts data in binary format and hashes it without further processing.

Apending an entry returns the index of the corresponding leaf (counting from one):

```
>>> tree.append_entry(b'foo')
1
>>> tree.append_entry(b'bar')
2
```

The index of a leaf can be used to retrieve the corresponding hash value:

```
>>> tree.get_leaf(1)
b'\x1d9\xfayq\xf4\xbf\x01\xa1\xc2\x0c\xb2\xa3\xfez\xf4he\xca\x9c\xd9\xb8@\xc2\x06=\xf8\
↪xfe\xc4\xffu'
>>>
>>> tree.get_leaf(2)
b'HY\x04\x12\x9b\xdd\xa5\xd1\xb5\xfb\xc6\xbcJ\x82\x95\x9e\xcf\xb9\x04-\xb4M\xc0\x8f\xe8~
↪6\x0b\n?%\x01'
```

Hash computation

Sometimes it is useful to be able to compute independently the hash value assigned to an data entry. For example, in order to verify the inclusion proof for an entry (see *below*) we need to know its hash value, which can be computed without querying the tree directly (provided that its binary format can be inferred according to some known contract).

To do so, we need to configure a standalone hasher that uses the same hash function as the tree and applies the same security policy:

```
from pymerkle.hasher import MerkleHasher

hasher = MerkleHasher(tree.algorithm, tree.security)
```

The commutation between index and entry is

```
assert tree.get_leaf(1) == hasher.hash_entry(b'foo')
```

Size

The *size* of the tree is the current number of leaves (i.e., data entries):

```
>>> tree.get_size()
5
```

It coincides with the index of the last appended leaf.

State

The *state* of the tree is uniquely determined by its current root-hash. This can be retrieved as follows:

```
>>> tree.get_state()
b'\xdcRj\xc4\x98\x81&}\x10\xf4<\x80\x8e\xc5\x92\xa1r\x08\xefxs<\xfa\x06""\xbeS[\xc70"'
```

The root-hash of any intermediate state can be retrieved by providing the corresponding size:

```
>>> tree.get_state(2)
b"9(jJU1b'Q\xd6\x84[\xb8\xef\xb4\xcf3\xbe\xc2\xc5\xf3\xf8C\ru\x84\x87Cq\xa3[\xda"
```

By convention, the empty tree state is the hash of the empty string:

```
>>> tree.get_state(0) == tree.hash_empty(b'')
True
```

Proofs

Pymerke is capable of generating proofs of *inclusion* and proofs of *consistency*. Both are modeled by the verifiable `MerkleProof` object.

Inclusion

Given any intermediate state, an inclusion proof is a path of hashes proving that a certain data entry has been appended at some previous moment and that the tree has not been afterwards tampered. Below the inclusion proof for the 3-rd entry against the state corresponding to the first 5 leaves:

```
proof = tree.prove_inclusion(3, 5)
```

The second argument is optional and defaults to the current tree size. Verification proceeds as follows:

```
from pymerkle import verify_inclusion

base = tree.get_leaf(3)
root = tree.get_state(5)

verify_inclusion(base, root, proof)
```

This checks that the path of hashes is indeed based on the acclaimed hash and that it resolves to the acclaimed state. Trying to verify against a forged base or state would raise an `InvalidProof` error:

```
>>> from pymerkle.hasher import MerkleHasher
>>>
>>> hasher = MerkleHasher(tree.algorithm, tree.security)
>>> forged = hasher.hash_raw(b'random')
>>>
>>> verify_inclusion(forged, root, proof)
Traceback (most recent call last):
...
pymerkle.proof.InvalidProof: Base hash does not match
>>>
>>> verify_inclusion(base, forged, proof)
Traceback (most recent call last):
...
pymerkle.proof.InvalidProof: State does not match
```

Consistency

Given any two intermediate states, a consistency proof is a path of hashes proving that the second is a valid later state of the first, i.e., that the tree has not been tampered with in the meanwhile. Below the consistency proof for the states with three and five leaves respectively:

```
proof = tree.prove_consistency(3, 5)
```

The second argument is optional and defaults to the current tree size. Verification proceeds as follows:

```
from pymerkle import verify_consistency

state1 = tree.get_state(3)
state2 = tree.get_state(5)

verify_consistency(state1, state2, proof)
```

This checks that an appropriate subpath of the included path of hashes resolves to the acclaimed prior state and the path of hashes as a whole resolves to the acclaimed later state. Trying to verify against forged states would raise an `InvalidProof` error:

```
>>> from pymerkle.hasher import MerkleHasher
>>>
>>> hasher = MerkleHasher(tree.algorithm, tree.security)
>>> forged = hasher.hash_raw(b'random')
>>>
>>> verify_consistency(forged, state2, proof)
Traceback (most recent call last):
...
pymerkle.proof.InvalidProof: Prior state does not match
>>>
>>> verify_consistency(state1, forged, proof)
Traceback (most recent call last):
...
pymerkle.proof.InvalidProof: Later state does not match
```

Serialization

A `MerkleProof` object can be serialized as follows:

```
data = proof.serialize()
```

This yields a JSON entity similar to this one:


```
{
  "metadata": {
      "algorithm": "sha256",
      "security": true,
      "size": 5
  },
  "rule": [
      0,
      1,
      0,
      0
  ],
  "subset": [],
  "path": [
      "4c79d0d62f7cf5ca8874155f2d3b875f2625da2bb3abc86bbd6833f25ba90e51",
      "5c7117fb9edb0cec387257891105da6a6616722af247083e2d6eda671529cdc5",
      "9531b48579f0e741979005d67ba64455a9f68b06630b3c431152d445ecd2716a",
      "bf36e59f88d0623d36dd3860e24a44fcc6bcd2ad88fdf67249dc1953f3605b51"
```


(continues on next page)

(continued from previous page)

```
  ]
}
```

The *metadata* section contains the parameters required for configuring the verification hasher (*algorithm* and *security*) along with the size of the state against which the proof was requested (*size*). The latter can be used in order to request the acclaimed state needed for proof verification (if not otherwise available). *Rule* determines parenthetization of hashes during path resolution and *subset* selects the hashes resolving to the acclaimed prior state (makes sense only for consistency proofs).

The verifiable proof-object can be retrieved as follows:

```python
from pymerkle import MerkleProof

proof = MerkleProof.deserialize(data)
```

1.6.2 Storage

Pymerkle is unopinionated on how leaves are appended to the tree, i.e., how entries should be stored in concrete. "Leaves" is an abstraction for the contiguously indexed data which the tree operates upon, no matter what their concrete form in persistent or volatile memory is. Specifying how to store entries and how to encode them (so that they become amenable to hashing operations) belongs to the particular application logic and amounts to implementing the internal storage interface presented in this section.

Interface

A Merkle-tree implementation is a concrete subclass of the `BaseMerkleTree` abstract base class. The latter encapsulates the cryptographic functionality in a storage agnostic fashion, i.e., without making assumptions about how entries are stored and accessed. It operates on top of an abstract storage interface, which is to be implemented by any concrete subclass:

```python
from pymerkle import BaseMerkleTree


class MerkleTree(BaseMerkleTree):

    def __init__(self, algorithm='sha256'):
        """
        Storage setup and superclass initialization
        """

    def _encode_entry(self, data):
        """
        Prepares data entry for hashing
        """

    def _store_leaf(self, data, digest):
        """
        Stores data hash in a new leaf and returns index
        """

    def _get_leaf(self, index):
```

(continues on next page)

(continued from previous page)

```
        """
        Returns the hash stored by the leaf specified
        """

    def _get_leaves(self, offset, width):
        """
        Returns hashes corresponding to the specified leaf range
        """

    def _get_size(self):
        """
        Returns the current number of leaves
        """
```

- `_encode_entry`: converts data entry to binary, so that it becomes amenable to hashing.
- `_store_leaf`: stores the output of hashing alogn with the original entry and returns the leaf index.
- `_get_leaf`: leaf hash by index (counting from one)
- `_get_leaves`: an iterable of the leaf hashes corresponding to the specified range
- `_get_size`: current tree size (number of leaves).

Various strategies are here possible. For example, data entry could be further processed by `_store_leaf` in order to conform to a given database schema and have the hash value stored in the appropriate table. Or, if a predefined schema is given that does not make space for hashes, the hash value could be forwarded to a dedicated datastore for future access; `_get_leaf` and `_get_leaves` would then have to access that separate datastore in order to make available the hash value.

Note: It is important to implement `_get_leaves` as efficiently as possible depending on your working framework. See *Optimizations* for details.

Here the exact interface to be implemented:

```
# pymerkle/base.py

class BaseMerkleTree(MerkleHasher, metaclass=ABCMeta):
   ...

   @abstractmethod
   def _encode_entry(self, data):
       """
       Should return the binary format of the provided data entry.

       :param data: data to encode
       :type data: whatever expected according to application logic
       :rtype: bytes
       """


   @abstractmethod
   def _store_leaf(self, data, digest):
       """
```

(continues on next page)

(continued from previous page)

```
        Should create a new leaf storing the provided data entry along with
        its hash value.

        :param data: data entry
        :type data: whatever expected according to application logic
        :param digest: hashed data
        :type digest: bytes
        :returns: index of newly appended leaf counting from one
        :rtype: int
        """


    @abstractmethod
    def _get_leaf(self, index):
        """
        Should return the hash stored at the specified leaf.

        :param index: leaf index counting from one
        :type index: int
        :rtype: bytes
        """


    @abstractmethod
    def _get_leaves(self, offset, width):
        """
        Should return in respective order the hashes stored by the leaves in
        the specified range.

        :param offset: starting position counting from zero
        :type offset: int
        :param width: number of leaves to consider
        :type width: int
        :rtype: iterable of bytes
        """


    @abstractmethod
    def _get_size(self):
        """
        Should return the current number of leaves

        :rtype: int
        """
    ...
```

Implementations

Pymerkle provides out of the box the following concrete implementations of `BaseMerkleTree`.

In memory

> **Warning:** This is a very memory inefficient implementation. Use it for debugging, testing and investigating the tree structure.

`InmemoryTree` is a non-persistent implementation where nodes reside in runtime.

```
from pymerkle import InmemoryTree

tree = InmemoryTree(algorithm='sha256')
```

Data is expected to be provided in binary:

```
index = tree.append_entry(b'foo')
```

It is hashed without further processing and can be accessed as follows:

```
data = tree.leaves[index - 1].entry
assert data == b'foo'
```

State coincides with the value of the current root-node:

```
assert tree.get_state() == tree.root.value
```

Nodes have a `right`, `left` and `parent` attribute, pointing to their right child, left child and parent node respectively. (Leaf nodes have no children, whereas the current root-node has no parent). These linkages allow for concrete path traversals. For example, the following loop detects the root-node starting from the first leaf of a non-empty tree:

```
leaf = tree.leaves[0]

curr = leaf
while curr.parent:
  curr = curr.parent

assert curr == tree.root
```

Concrete path traversals are used under the hood for visualizing the tree by means of printing:

```
>>> print(tree)

 └─346ec544...
     ├──bbe0bdaf...
     │    ├──39286a4a...
     │    │    ├──1d2039fa...
     │    │    └──48590412...
     │    └──0bf15c4f...
     │         ├──b06d6958...
     │         └──5a43bc14...
     │
```

(continues on next page)

(continued from previous page)

```
    └─4c715fb1...
        ├──7a4b8eff...
        │   ├──2e219794...
        │   └──1c0c3f26...
        └──e9345fea...
            ├──2c3bb97e...
            └──dcd08bea...
```

Sqlite

`SqliteTree` uses a SQLite database to persistently store entries. It is a wrapper of sqlite3, suitable for leightweight applications that do not require separate server processes for the database.

```
from pymerkle import SqliteTree

tree = SqliteTree('merkle.db')
```

This opens a connection to the provided database, which will also be created if not already existent.

Note: The database schema consists of a single table called *leaf* with two columns: *index*, which is the primary key serving as leaf index, and *entry*, which is a blob field storing the appended data.

Data is expected to be provided in binary:

```
index = tree.append_entry(b'foo')
```

It is hashed without further processing and can be accessed as follows:

```
data = tree.get_entry(index)
assert data == b'foo'
```

In order to efficiently append multiple entries at once, you can do the following:

```
entries = [f'entry-{i + 1}'.encode() for i in range(100000)]

tree.append_entries(entries, chunksize=1024)
```

where `chunksize` controls the number of insertions per database transaction (defaults to 100,000).

It is suggested to close the connection to the database when ready:

```
tree.con.close()
```

Alternatively, initialize the tree as context-manager to ensure that this will be done without taking explicit care:

```
with SqliteTree('merkle.db') as tree:
    ...
```

Examples

Warning: The following exaples are only for the purpose of reference and understanding

Simple list

This is a simple non-persistent implementation utilizing a list as storage. It expects entries to be strings, which it encodes in utf-8 before hashing.

```
from pymerkle import BaseMerkleTree


class MerkleTree(BaseMerkleTree):

    def __init__(self, algorithm='sha256'):
        self.hashes = []

        super().__init__(algorithm)


    def _encode_entry(self, data):
        return data.encode('utf-8')


    def _store_leaf(self, data, digest):
        self.hashes += [digest]

        return len(self.hashes)


    def _get_leaf(self, index):
        value = self.hashes[index - 1]

        return value


    def _get_leaves(self, offset, width):
        values = self.hashes[offset: offset + width]

        return values


    def _get_size(self):
        return len(self.hashes)
```

Unix DBM

This is a hasty implementing using dbm to persistently store entries in a "merkledb" file. It expects strings as entries and encodes them in utf-8 before hashing.

```python
import dbm
from pymerkle import BaseMerkleTree


class MerkleTree(BaseMerkleTree):

    def __init__(self, algorithm='sha256'):
        self.dbfile = 'merkledb'
        self.mode = 0o666

        with dbm.open(self.dbfile, 'c', mode=self.mode) as db:
            pass

        super().__init__(algorithm)


    def _encode_entry(self, data):
        return data.encode('utf-8')


    def _store_leaf(self, data, digest):
        with dbm.open(self.dbfile, 'w', mode=self.mode) as db:
            index = len(db) + 1
            db[hex(index)] = b'|'.join(data, digest)

        return index


    def _get_leaf(self, index):
        with dbm.open(self.dbfile, 'r', mode=self.mode) as db:
            value = db[hex(index)].split(b'|')[1]

        return value


    def _get_leaves(self, offset, width):
        values = []
        with dbm.open(self.dbfile, 'r', mode=self.mode) as db:
            for index in range(offset + 1, width + 1):
                value = db[hex(index)].split(b'|')[index]
                values += [value]

        return value


    def _get_size(self):
        with dbm.open(self.dbfile, 'r', mode=self.mode) as db:
            size = len(db)
```

(continues on next page)

(continued from previous page)

```
        return size
```

Django app

1.6.3 Optimizations

Interior nodes are not assumed to be stored anywhere. The tree structure is determined by the function which computes root-hashes for arbitrary leaf ranges on the fly. The performance of the tree depends highly on the efficiency of this operation. The recursive version of this function (e.g., RFC 9162, Section 2) is slow, affecting significantly state computation and generation of proofs.

Subroots

The above operation can be made iterative by accumulatively hashing together the root-hashes for ranges whose size is a power of two ("subroots") and can as such be computed efficiently. Subroot computation has significant impact on performance (>500% speedup) while keeping peak memory usage reasonably low (e.g., 200 MiB for a tree with several tens of millions of entries) and linear with respect to tree size.

Note: For, say, comparison purposes, you can disable this feature by passing `disable_optimizations=True` when initializing the `BaseMerkleTree` superclass.

Effect of I/O operation

Subroot computation is CPU-bound except for loading leaf hashes to memory. This operation is implementation specific, since it depends on the particular storage backend which the tree operates upon (see `_get_leaves` in *this* section). The effect of this operation (usually I/O) can be significant. Take care to implement it in the most efficient way facilitated by your working framework (e.g., bulk fetching the dataset).

Caching

In view of the above technique, subroot computation is the only massively repeated and relatively costly operation. It thus makes sense to apply memoization for ranges whose size exceeds a certain threshold (128 leaves by default). For example, after sufficiently many cache hits (e.g. 2MiB cache memory), proof generation becomes at least 5 times faster for a tree with several tens of million of entries. Practically, a pretty big tree with sufficiently long uptime will respond instantly with negligible penalty in memory usage.

Cache capacity is controlled in bytes via the `capacity` parameter, which is passed to `BaseMerkleTree` and defaults to 1GiB (this should be overabundant for any imaginable use case). The minimum size of leaf ranges with cacheable root-hash is controlled via the `threshold` parameter, which is similarly passed to `BaseMerkleTree` and defaults to 128.

Note: For, say, comparison purposes, you can disable this feature by passing `disable_cache=True` when initializing the `BaseMerkleTree` superclass.

1.6.4 pymerkle

pymerkle package

Subpackages

pymerkle.concrete package

Submodules

pymerkle.concrete.inmemory module

pymerkle.concrete.sqlite module

Submodules

pymerkle.constants module

pymerkle.core module

pymerkle.hasher module

pymerkle.proof module

pymerkle.utils module

1.7 Indices and tables

- genindex
- modindex
- search