seccs Documentation

Release 0.0.5

Author

Oct 02, 2017

Contents

seccs is a Python library that realizes a secure and efficient hash-table-like data structure for contents on top of any existing key-value store as provided by, e.g., cloud storage providers.

It has been developed as part of the work *[LS17]* at CISPA, Saarland University.

Contents:

CHAPTER 1

Installation

Run:

```
$ pip install seccs
```

If you want to use AES-SIV encryption (you probably want!), you also need to install PyCrypto 2.7a1 which is not yet available in PyPI:

```
$ pip install https://ftp.dlitz.net/pub/dlitz/crypto/pycrypto/pycrypto-2.7a1.tar.gz
```

CHAPTER 2

Usage and Overview

seccs is a Python implementation of *sec-cs*, a secure and efficient hash-table-like data structure for contents. It stores its data on top of any existing database providing a key-value store interface. Thus, it is likewise usable with in-memory `dict` objects, persistent databases like `ZODB`, and many cloud storage providers.

Its details are described in *[LS17]*. In short, it is suitable for usage on *untrusted* cloud storage and has the following desirable properties:

- **Confidentiality:** Stored contents are securely encrypted using a symmetric key.
- **Authenticity:** *sec-cs* guarantees authenticity of all stored contents, irrespective of gurantees of the underlying database.
- **Storage Efficiency:** Data deduplication strategies are applied to all stored contents. When storing new contents, overlapping parts of existing contents are automatically reused as to avoid redundancy. *sec-cs* is optimized for efficiency in presence of *many* similar contents: Storage costs of an n-bytes content that differs only slightly from an existing content are in O(log n).

Typical Use Case

In the most-typical configuration, *sec-cs* chunks its contents hierarchically using ML-CDC (see *[LS17]*), usually relying on Rabin Karp hashes, and stores the resulting nodes in a *database* after applying AES-SIV-256 for encryption and authentication. From a user perspective, we have to initialize a suitable database object and a 32-bytes key first.

Database and key setup:

```
>>> database = dict()
>>> import os
>>> key = os.urandom(32)
```

Note that we might want to store the database and the key at some persistent location in practice.

Next, we need to create a *crypto wrapper* which is in charge of all the cryptographic operations. Depending on our security goals (e.g., whether encryption is required), we could choose any suitable wrapper from *seccs.crypto_wrapper*. Afterwards, we can instantiate the data structure.

Choice of *crypto wrapper* and instantiation of data structure:

```
>>> import seccs
>>> crypto_wrapper = seccs.crypto_wrapper.AES_SIV_256(key)  # install PyCrypto>=2.
↪7a1 to use AES-SIV
>>> seccs = seccs.SecCSLite(256, database, crypto_wrapper)  # 256 is the chunk␣
↪size
```

Note: Internally, *sec-cs* splits contents into chunks, creates a tree of chunks for each of them and inserts each node separately into the *database*. The first parameter specifies the desired *average* size of nodes inserted into the database. As deduplication is performed at the chunk level, large chunk sizes decrease deduplication performance, but they also create less storage overhead when storing non-deduplicable contents as fewer nodes have to be stored.

Performance is discussed in detail in *[LS17]*. If high redundancy is expected, 256 bytes is typically a good compromise; otherwise, larger chunk sizes might be more suitable.

We can now insert contents...

```
>>> content = "This is a test content."
>>> digest = seccs.put_content(content)
>>> repr(digest)
'\x08,f+\xa74\xdc\x0f\xe5Oo\xcb;\x83\xb9T\x00\x00\x00\x00\x00\x00\x00\x17'
```

...retrieve them...

```
>>> seccs.get_content(digest)
This is a test content.
```

...and delete them as soon as they are not needed anymore:

```
>>> seccs.delete_content(digest)
```

Storage Efficiency

seccs avoids redundancy in the *database* wherever possible, as gets clear in the following example.

Consider this function for measuring the *database*'s current storage costs in bytes:

```
>>> import sys
>>> def dbsize(db):
>>>     return sum([sys.getsizeof(k) + sys.getsizeof(v) for (k, v) in db.items()])
```

Initially, the database is empty:

```
>>> dbsize(database)
0
```

Insertion of a 1 MiB content clearly causes some storage costs:

```
>>> content1 = os.urandom(1024*1024)
>>> digest1 = seccs.put_content(content1)
>>> dbsize(database)
1583030
```

But inserting the same content for a second time does not incur additional costs:

```
>>> content2 = content1
>>> digest2 = seccs.put_content(content2)
>>> digest1 == digest2  # identical contents yield identical digests
True
>>> dbsize(database)
1583030
```

Clearly, the database grows if different contents are inserted. However, these costs are low if inserted contents are similar to existing ones.

Only about 2.3 KiB are required to store another 1 MiB content with one byte changed:

```
>>> content3 = ''.join([content1[:512*1024], 'x', content1[512*1024+1:]])
>>> digest3 = seccs.put_content(content3)
>>> dbsize(database)
1585395
```

Costs are similar even if the identical parts are shifted...

```
>>> content4 = ''.join([content1[:512*1024], 'xyz', content1[512*1024+1:]])
>>> digest4 = seccs.put_content(content4)
>>> dbsize(database)
1588010
```

...and deduplication is also performed if a content consists of parts of different existing contents:

```
>>> content5 = ''.join([content1, content3, content4])
>>> digest5 = seccs.put_content(content5)
>>> dbsize(database)
1591009
```

In the last example, the growth was about 3 KiB.

Furthermore, storage space is reclaimed completely when contents are removed:

```
>>> seccs.delete_content(digest5)
>>> seccs.delete_content(digest4)
>>> seccs.delete_content(digest3)
>>> seccs.delete_content(digest2)
>>> dbsize(database)
1583030
>>> seccs.delete_content(digest1)
>>> dbsize(database)
0
```

Note: Every `seccs.delete_content()` call undos eactly one `seccs.put_content()` call. Thus, even if the same content has been inserted twice, yielding only a single digest, it has to be deleted twice as well to get actually removed.

References:

CHAPTER 3

seccs package

Module contents

sec-cs — the SECure Content Store.

This module provides an implementation of the secure content store data structure introduced in *[LS16]*.

sec-cs allows secure and efficient storage of contents in an existing key-value database, providing the following features:

- **Confidentiality:** Stored contents are securely encrypted using a symmetric key.
- **Authenticity:** *sec-cs* guarantees authenticity of all stored contents, irrespective of gurantees of the underlying database.
- **Storage Efficiency:** Data deduplication strategies are applied to all stored contents. When storing new contents, overlapping parts of existing contents are automatically reused as to avoid redundancy. Storage costs of an n-bytes content that differs only slightly from an existing content are in O(log n).

Note: The only *sec-cs* implementation currently included in this module is called `SecCSLite`. While it is likely suitable for many projects and can be used as is, it is actually intended as a base class for a much more powerful variant, `SecCS`, which makes some slight changes to the internal storage structure and will be published in the near future.

References

class `seccs.`**`SecCSLite`**(*chunk_size*, *database*, *crypto_wrapper*, *chunking_strategy=None*, *reference_counter=None*, ***kwargs*)

Bases: `object`

Secure Content Store *lite*.

Basic implementation of the Secure Content Store data structure, supports only insertion (put), retrieval (get) and deletion (delete) of contents.

Parameters

- **chunk_size** (*int*) – Target chunk size, i.e., expected size of all chunks stored in the database.
- **database** – Persistent database used as backend. Can be any object with a dict-like interface, i.e., any object implementing the operations __getitem__, __setitem__, __delitem__ and __contains__.
- **crypto_wrapper** (*crypto_wrapper.BaseCryptoWrapper*) – Crypto wrapper object that specifies cryptographic operations to be applied to data stored in the *database*.
- **chunking_strategy** (Optional[`fastchunking.BaseChunkingStrategy`]) – Chunking strategy that shall be applied to contents. Defaults to Rabin-Karp-based content defined chunking with 48-bytes window size.
- **reference_counter** (Optional[*seccs.rc.BaseReferenceCounter*]) – Reference counting strategy. By default, reference counters are stored in *database* under keys *key* || *"r"*, where *key* is the key whose references are counted.
- ****kwargs** – Extra keyword arguments that you should NOT use unless you really, really know what you are doing, e.g.:
  - length_to_height_fn: Function that resolves content lengths to appropriate chunk tree heights. May be used to modify the multi-level chunking approach performed by default, e.g., to degrade it to single-level chunking or similar.
  - height_to_chunk_size_fn: Function that computes the target chunk size for a specific level (height) of a chunk tree. May be used to create *imbalanced* chunk trees.

Raises *seccs.UnsupportedChunkSizeError* – If the chosen chunk size would create superchunk nodes with less than two expected children as efficiency guarantees would fail in this case (see *[LS16]*).

delete_content(*k*, *ignore_rc=False*)

Delete a content from the data structure.

Decreases the content's reference counter and deletes its root chunk (possibly including children) if no references are left.

Parameters

- **k** (*str*) – The digest under which the content is stored.
- **ignore_rc** (*Optional[bool]*) – If True, decrease of reference counter of the root node is skipped and root node (possibly including children) is deleted straight away.

get_content(*k*)

Retrieve a content from the data structure.

Parameters **k** (*str*) – The digest under which the content is stored.

Returns The content bytestring.

Return type str

put_content(*m*, *ignore_rc=False*)

Insert a content into the data structure.

Parameters

- **m** (*str*) – The message or content that shall be processed and inserted into the data structure.
- **ignore_rc** (*Optional[bool]*) – If True, increase of reference counter for the root node of the generated chunk tree is skipped. Defaults to False.

Returns Digest of the content that allows its retrieval using *get_content()*.

Return type str

put_content_and_check_if_new(*m*, *ignore_rc=False*)

Insert a content into the data structure.

Like *put_content()*, but return value includes information whether the content had been in the data structure before.

Parameters

- **m** (*str*) – The message or content that shall be processed and inserted into the data structure.
- **ignore_rc** (*Optional[bool]*) – If True, increase of reference counter for the root node of the generated chunk tree is skipped. Defaults to False.

Returns (digest, is_new), where *digest* is the content's digest that allows its retrieval using *get_content()*, and *is_new* is True if the content has been inserted for the first time and False if it had existed before.

Return type tuple

exception seccs.**UnsupportedChunkSizeError**

Bases: exceptions.Exception

Raised when trying to instantiate SecCS with an unsupported chunk size.

Submodules

seccs.crypto_wrapper module

Crypto wrapper implementations.

A crypto wrapper encapsulates all cryptographic operations that have to be applied to node representations before they can be safely inserted into the (untrusted) database, e.g., encryption and authentication. It defines the keys (digests) under which (wrapped) node representations are stored and specifies how node representations can be extracted from (digest, value) pairs stored in the database.

This module includes several crypto wrapper implementations with different security properties.

class seccs.crypto_wrapper.**AES_SIV_256**(*key*)

Bases: *seccs.crypto_wrapper.BaseCryptoWrapper*

AES-SIV-256 crypto wrapper.

Provides confidentiality and authenticity for chunk tree nodes based on a symmetric 32-bytes key specified during instantiation.

Root nodes are handled identically to inner nodes at the same level.

Nodes are represented as follows:

- value: AES-SIV-256(<key>, <value>, additional_data=<height>)

- digest: <digest produced by AES-SIV-256>

Parameters **key** (*str*) – Cryptographic key used for symmetric encryption and authentication.

Note: Requires PyCrypto >= 2.7a1.

---

unwrap_value(*value*, *digest*, *height*, *is_root*, *length=-1*)

Decrypts and verifies node representation and returns the result on success.

Raises *AuthenticityError* – If digest does not match.

See *BaseCryptoWrapper.unwrap_value()*.

wrap_value(*value*, *height*, *is_root*)

Encrypts node representation using deterministic authenticated encryption, i.e., with AES in SIV mode, including node height as additional data that is authenticated, resulting in a digest (MAC) that is used as *digest* and a ciphertext that is used as *value*.

Note: As AES-SIV cannot encrypt empty contents, a distinguished zero digest is artifically assigned to empty node representations instead.

See *BaseCryptoWrapper.wrap_value()*.

class seccs.crypto_wrapper.**AES_SIV_256_DISTINGUISHED_ROOT**(*key*)

Bases: *seccs.crypto_wrapper.AES_SIV_256*

AES-SIV-256 crypto wrapper with distinguished root representation.

Provides confidentiality and authenticity for chunk tree nodes based on a symmetric 32-bytes key specified during instantiation.

Root nodes are handled differently from inner nodes at the same level.

Nodes are represented as follows:

- value: AES-SIV-256(<key>, <value>, additional_data=<height>||<is_root>)
- digest: <digest produced by AES-SIV-256>

Parameters **key** (*str*) – Cryptographic key used for symmetric encryption and authentication.

Note: Requires PyCrypto >= 2.7a1.

---

unwrap_value(*value*, *digest*, *height*, *is_root*, *length=-1*)

Decrypts and verifies node representation and returns the result on success.

Raises *AuthenticityError* – If digest does not match.

See *BaseCryptoWrapper.unwrap_value()*.

wrap_value(*value*, *height*, *is_root*)

Encrypts node representation using deterministic authenticated encryption, i.e., with AES in SIV mode, including node height and is_root flag as additional data that is authenticated, resulting in a digest (MAC) that is used as *digest* and a ciphertext that is used as *value*.

See *AES_SIV_256.wrap_value()*.

exception seccs.crypto_wrapper.**AuthenticityError**
Bases: *seccs.crypto_wrapper.IntegrityError*

Raised if an authenticity verification fails.

class seccs.crypto_wrapper.**BaseCryptoWrapper**
Bases: object

Abstract class specifying the crypto wrapper interface.

unwrap_value(*value*, *digest*, *height*, *is_root*, *length=-1*)
Converts the representation of a node as stored in the database (e.g., an encrypted representation) into its *normal*, e.g., decrypted, representation.

Parameters

- **value** (*str*) – Wrapped node representation.
- **digest** (*str*) – Key under which the node is stored in the database.
- **height** (*int*) – Height of the node in its chunk tree.
- **is_root** (*bool*) – Whether or not the node is the chunk tree's root.
- **[Optional** (*length*) – Length of the content represented by this node. Defaults to -1.

Returns Node representation.

Return type str

wrap_value(*value*, *height*, *is_root*)
Converts a node representation into a (digest, value) pair that can be safely inserted into the database.

Parameters

- **value** (*str*) – The node representation.
- **height** (*int*) – The height of the node in its chunk tree.
- **is_root** (*bool*) – Whether or not the node is the chunk tree's root.

Returns (wrapped_value, digest).

Return type tuple

class seccs.crypto_wrapper.**HMAC_SHA_256**(*key*)
Bases: *seccs.crypto_wrapper.BaseCryptoWrapper*

HMAC-SHA-256 crypto wrapper.

Provides authenticity for chunk tree nodes based on a symmetric 32-bytes key specified during instantiation.

Root nodes are handled identically to inner nodes at the same level.

Nodes are represented as follows:

- value: <value>
- digest: HMAC-SHA-256(<key>, <height> || <value>)

Parameters **key** (*str*) – Cryptographic key used for symmetric authentication.

unwrap_value(*value*, *digest*, *height*, *is_root*, *length=-1*)
Verifies SHA-256-based HMAC and returns node representation on success.

Raises *AuthenticityError* – If digest does not match.

See *BaseCryptoWrapper.unwrap_value()*.

wrap_value(*value*, *height*, *is_root*)
Uses SHA-256-based HMAC of node representation and height as digest and value as is.

See *BaseCryptoWrapper.wrap_value()*.

class seccs.crypto_wrapper.**HMAC_SHA_256_DISTINGUISHED_ROOT**(*key*)
Bases: *seccs.crypto_wrapper.HMAC_SHA_256*

HMAC-SHA-256 crypto wrapper with distinguished root representation.

Provides authenticity for chunk tree nodes based on a symmetric 32-bytes key specified during instantiation.

Root nodes are handled differently from inner nodes at the same level.

Nodes are represented as follows:

- value: <value>
- digest: HMAC-SHA-256(<key>, <height> || <is_root> || <value>)

Parameters **key** (*str*) – Cryptographic key used for symmetric authentication.

unwrap_value(*value*, *digest*, *height*, *is_root*, *length=-1*)
Verifies SHA-256-based HMAC and returns node representation on success.

Raises *AuthenticityError* – If digest does not match.

See *BaseCryptoWrapper.unwrap_value()*.

wrap_value(*value*, *height*, *is_root*)
Uses SHA-256-based HMAC of node representation, height and is_root flag as digest and value as is.

See *BaseCryptoWrapper.wrap_value()*.

exception seccs.crypto_wrapper.**IntegrityError**
Bases: exceptions.ValueError

Raised if an integrity verification fails.

class seccs.crypto_wrapper.**SHA_256**
Bases: *seccs.crypto_wrapper.BaseCryptoWrapper*

SHA-256 crypto wrapper.

Provides integrity for chunk tree nodes.

Nodes are represented as follows:

- value: <value>
- digest: SHA-256(<value>)

unwrap_value(*value*, *digest*, *height*, *is_root*, *length=-1*)
Verifies SHA-256 hash and returns node representation on success.

Raises *IntegrityError* – If digest does not match.

See *BaseCryptoWrapper.unwrap_value()*.

wrap_value(*value*, *height*, *is_root*)
Uses SHA-256 hash of node representation as digest and value as is.

See *BaseCryptoWrapper.wrap_value()*.

seccs.rc module

Simple reference counter implementations.

class seccs.rc.**BaseReferenceCounter**
Bases: object

Abstract base class for reference counters.

dec(*key*)
Abstract decrement interface.

Parameters **key** – Key whose reference counter shall be decremented.

Returns Number of references of key after decrement.

get(*key*)
Abstract get interface.

Parameters **key** – Key whose reference counter shall be retrieved.

Returns Number of references of key.

inc(*key*)
Abstract increment interface.

Parameters **key** – Key whose reference counter shall be incremented.

Returns Number of references of key after increment.

class seccs.rc.**DatabaseReferenceCounter**(*database*)
Bases: *seccs.rc.BaseReferenceCounter*

Database-backed reference counter.

Uses a given database to store reference counters. The reference counter of an element *key* is stored as follows:

- If its value is 0, *key* is not stored in the database.
- If its value is > 0, its int value is stored in the database under *key*.

Parameters **database** – Database object with a dict-like interface, i.e., implementing the operations __getitem__, __setitem__ and __delitem__.

dec(*key*)
Decrementes reference counter of key.

See *BaseReferenceCounter.dec()*.

get(*key*)
Gets reference counter of key.

See *BaseReferenceCounter.get()*.

inc(*key*)
Incrementes reference counter of key.

See *BaseReferenceCounter.inc()*.

class seccs.rc.**KeySuffixDatabaseReferenceCounter**(*database*, *suffix*)
Bases: *seccs.rc.DatabaseReferenceCounter*

Database-backed reference counter.

Similar to *DatabaseReferenceCounter*, but the reference counter of a *key* is not stored directly under *key*, but under *key* || *suffix*.

Parameters

- **database** – Database object with a dict-like interface, i.e., implementing the operations __getitem__, __setitem__ and __delitem__.
- **suffix** – Suffix for keys.

dec(*key*)
Decrementes reference counter of key.

See *DatabaseReferenceCounter.dec()*.

get(*key*)
Gets reference counter of key.

See *DatabaseReferenceCounter.get()*.

inc(*key*)
Incrementes reference counter of key.

See *DatabaseReferenceCounter.inc()*.

class seccs.rc.**NoReferenceCounter**
Bases: *seccs.rc.BaseReferenceCounter*

Non-counting reference counter, always returns 1 for any key.

Can be used to disable reference counting where a reference counter is required.

dec(*key*)
Decrement interface.

Returns 1

get(*key*)
Get interface.

Returns 1

inc(*key*)
Increment interface.

Returns 1

CHAPTER 4

Testing

seccs uses tox for testing, so simply run:

```
$ tox
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[LS17] Dominik Leibenger and Christoph Sorge (2017). sec-cs: Getting the Most out of Untrusted Cloud Storage. In Proceedings of the 42nd IEEE Conference on Local Computer Networks (LCN 2017), 2017. (Preprint: arXiv:1606.03368)

[LS16] Dominik Leibenger and Christoph Sorge (2016). sec-cs: Getting the Most out of Untrusted Cloud Storage. arXiv:1606.03368

Python Module Index

S

Index