fastchunking Documentation

Release 0.0.4

Author

Apr 20, 2018

Contents

fastchunking is a Python library that contains efficient and easy-to-use implementations of string chunking algorithms. It has been developed as part of the work *[LS17]* at CISPA, Saarland University.

CHAPTER 1

Contents

1.1 Installation

Run:

```
$ pip install fastchunking
```

Note: For performance reasons, parts of this library are implemented in C++. Installation from a source distribution, thus, requires availability of a correctly configured C++ compiler.

1.2 Usage and Overview

fastchunking provides efficient implementations for different string chunking algorithms, e.g., static chunking (SC) and content-defined chunking (CDC).

1.2.1 Static Chunking (SC)

Static chunking splits a message into fixed-size chunks.

Let us consider a random example message that shall be chunked:

```
>>> import os
>>> message = os.urandom(1024*1024)
```

Static chunking is trivial when chunking a single message:

```
>>> import fastchunking
>>> sc = fastchunking.SC()
>>> chunker = sc.create_chunker(chunk_size=4096)
```

```
>>> chunker.next_chunk_boundaries(message)
[4096, 8192, 12288, ...]
```

A large message can also be chunked in fragments, though:

```
>>> chunker = sc.create_chunker(chunk_size=4096)
>>> chunker.next_chunk_boundaries(message[:10240])
[4096, 8192]
>>> chunker.next_chunk_boundaries(message[10240:])
[2048, 6144, 10240, ...]
```

1.2.2 Content-Defined Chunking (CDC)

fastchunking supports content-defined chunking, i.e., chunking of messages into fragments of variable lengths.

Currently, a chunking strategy based on Rabin-Karp rolling hashes is supported.

As a rolling hash computation on plain-Python strings is incredibly slow with any interpreter, most of the computation is performed by a C++ extension which is based on the *ngramhashing* library by Daniel Lemire, see: https://github.com/lemire/rollinghashcpp

Let us consider a random message that should be chunked:

```
>>> import os
>>> message = os.urandom(1024*1024)
```

When using static chunking, we have to specify a rolling hash window size (here: 48 bytes) and an optional seed value that affects the pseudo-random distribution of the generated chunk boundaries.

Despite that, usage is similar to static chunking:

```
>>> import fastchunking
>>> cdc = fastchunking.RabinKarpCDC(window_size=48, seed=0)
>>> chunker = cdc.create_chunker(chunk_size=4096)
>>> chunker.next_chunk_boundaries(message)
[7475L, 10451L, 12253L, 13880L, 15329L, 19808L, ...]
```

Chunking in fragments is straightforward:

```
>>> chunker = cdc.create_chunker(chunk_size=4096)
>>> chunker.next_chunk_boundaries(message[:10240])
[7475L]
>>> chunker.next_chunk_boundaries(message[10240:])
[211L, 2013L, 3640L, 5089L, 9568L, ...]
```

1.2.3 Multi-Level Chunking (ML-*)

Multiple chunkers of the same type (but with different chunk sizes) can be efficiently used in parallel, e.g., to perform multi-level chunking *[LS17]*.

Again, let us consider a random message that should be chunked:

```
>>> import os
>>> message = os.urandom(1024*1024)
```

Usage of multi-level-chunking, e.g., ML-CDC, is easy:

```
>>> import fastchunking
>>> cdc = fastchunking.RabinKarpCDC(window_size=48, seed=0)
>>> chunk_sizes = [1024, 2048, 4096]
>>> chunker = cdc.create_multilevel_chunker(chunk_sizes)
>>> chunker.next_chunk_boundaries_with_levels(message)
[(1049L, 2L), (1511L, 1L), (1893L, 2L), (2880L, 1L), (2886L, 0L),
(3701L, 0L), (4617L, 0L), (5809L, 2L), (5843L, 0L), ...]
```

The second value in each tuple indicates the highest chunk size that leads to a boundary. Here, the first boundary is a boundary created by the chunker with index 2, i.e., the chunker with 4096 bytes target chunk size.

Note: Only the highest index is output if multiple chunkers yield the same boundary.

Warning: Chunk sizes have to be passed in correct order, i.e., from lowest to highest value.

References:

1.3 fastchunking package

1.4 Performance

Computation costs for *static chunking* are barely measurable: As chunking does not depend on the actual message but only its length, computation costs are essentially limited to a single `xrange` call.

Content-defined chunking, however, is expensive: The algorithm has to compute hash values for rolling hash window contents at *every* byte position of the message that is to be chunked. To minimize costs, fastchunking works as follows:

1. The message (fragment) is passed in its entirety to the C++ extension.
2. Chunking is performed within the C++ extension.
3. The resulting list of chunk boundaries is communicated back to Python and converted into a Python list.

Based on a 100 MiB random content, the author measured the following throughput on an Intel Core i7-4600U in a single, non-representative test run:

chunk size	throughput
64 bytes	49 MiB/s
128 bytes	57 MiB/s
256 bytes	62 MiB/s
512 bytes	63 MiB/s
1024 bytes	67 MiB/s
2048 bytes	68 MiB/s
4096 bytes	70 MiB/s
8192 bytes	71 MiB/s
16384 bytes	71 MiB/s
32768 bytes	71 MiB/s

1.5 Testing

fastchunking uses tox for testing, so simply run:

```
$ tox
```

CHAPTER 2

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)