tamp

Release 0.0.0

Brian Pugh

May 09, 2024

CONTENTS:

Documentation: https://tamp.readthedocs.io/en/latest/

Source Code: https://github.com/BrianPugh/tamp

Tamp is a low-memory, DEFLATE-inspired lossless compression library intended for embedded targets.

Tamp delivers the highest data compression ratios, while using the least amount of RAM and firmware storage.

CHAPTER

ONE

FEATURES

- Various language implementations available:
  - Pure Python reference:
    - `tamp/__init__.py`, `tamp/compressor.py`, `tamp/decompressor.py`
    - `pip install tamp` will use a python-bound C implementation optimized for speed.
  - Micropython:
    - Native Module (suggested micropython implementation).
      - `mpy_bindings/`
    - Viper.
      - `tamp/__init__.py`, `tamp/compressor_viper.py`, `tamp/decompressor_viper.py`
  - C library:
    - `tamp/_c_src/`
- High compression ratios, low memory use, and fast.
- Compact compression and decompression implementations.
  - Compiled C library is <4KB (compressor + decompressor).
- Mid-stream flushing.
  - Allows for submission of messages while continuing to compress subsequent data.
- Customizable dictionary for greater compression of small messages.
- Convenient CLI interface.

CHAPTER

TWO

INSTALLATION

Tamp contains 4 implementations:

1. A reference desktop CPython implementation that is optimized for readability (and **not** speed).
2. A Micropython Native Module implementation (fast).
3. A Micropython Viper implementation (not recommended, please use Native Module).
4. A C implementation (with python bindings) for accelerated desktop use and to be used in C projects (very fast).

This section instructs how to install each implementation.

2.1 Desktop Python

The Tamp library and CLI requires Python `>=3.8` and can be installed via:

```
pip install tamp
```

2.2 MicroPython

2.2.1 MicroPython Native Module

Tamp provides pre-compiled [native modules]{.title-ref} that are easy to install, are small, and are incredibly fast.

Download the appropriate `.mpy` file from the release page.

- Match the micropython version.
- Match the architecture to the microcontroller (e.g. `armv6m` for a pi pico).

Rename the file to `tamp.mpy` and transfer it to your board. If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = "https://github.com/BrianPugh/tamp/releases/download/v1.4.0/tamp-1.4.0-mpy1.22-
↪armv6m.mpy"
```

2.2.2 MicroPython Viper

NOT RECOMMENDED, PLEASE USE NATIVE MODULE

For micropython use, there are 3 main files:

1. `tamp/__init__.py` - Always required.
2. `tamp/decompressor_viper.py` - Required for on-device decompression.
3. `tamp/compressor_viper.py` - Required for on-device compression.

For example, if on-device decompression isn't used, then do not include `decompressor_viper.py`. If manually installing, just copy these files to your microcontroller's `/lib/tamp` folder.

If using mip, tamp can be installed by specifying the appropriate `package-*.json` file.

```
mip install github:brianpugh/tamp  # Defaults to package.json: Compressor & Decompressor
mip install github:brianpugh/tamp/package-compressor.json  # Compressor only
mip install github:brianpugh/tamp/package-decompressor.json  # Decompressor only
```

If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = [
    "https://github.com/BrianPugh/tamp/blob/main/tamp/__init__.py",
    "https://github.com/BrianPugh/tamp/blob/main/tamp/compressor_viper.py",
    "https://github.com/BrianPugh/tamp/blob/main/tamp/decompressor_viper.py",
]
```

2.3 C

Copy the `tamp/_c_src/tamp` folder into your project. For more information, see the documentation.

CHAPTER

THREE

USAGE

Tamp works on desktop python and micropython. On desktop, Tamp is bundled with the `tamp` command line tool for compressing and decompressing tamp files.

3.1 CLI

3.1.1 Compression

Use `tamp compress` to compress a file or stream. If no input file is specified, data from stdin will be read. If no output is specified, the compressed output stream will be written to stdout.

```
$ tamp compress --help
Usage: tamp compress [ARGS] [OPTIONS]

Compress an input file or stream.

─ Parameters␣
↪──────────────────────────────────────────────────────────────────────────────
│ INPUT,--input    -i  Input file to compress. Defaults to stdin.                 ␣
↪    │
│ OUTPUT,--output  -o  Output compressed file. Defaults to stdout.                ␣
↪    │
│ --window         -w  Number of bits used to represent the dictionary window. [default:␣
↪10] │
│ --literal        -l  Number of bits used to represent a literal. [default: 8]   ␣
↪    │
──────────────────────────────────────────────────────────────────────────────
```

Example usage:

```
tamp compress enwik8 -o enwik8.tamp  # Compress a file
echo "hello world" | tamp compress | wc -c  # Compress a stream and print the compressed␣
↪size.
```

The following options can impact compression ratios and memory usage:

- `window` - `2^window` plaintext bytes to look back to try and find a pattern. A larger window size will increase the chance of finding a longer pattern match, but will use more memory, increase compression time, and cause each pattern-token to take up more space. Try smaller window values if compressing highly repetitive data, or short messages.

- `literal` - Number of bits used in each plaintext byte. For example, if all input data is 7-bit ASCII, then setting this to 7 will improve literal compression ratios by 11.1%. The default, 8-bits, can encode any binary data.

3.1.2 Decompression

Use `tamp decompress` to decompress a file or stream. If no input file is specified, data from stdin will be read. If no output is specified, the compressed output stream will be written to stdout.

```
$ tamp decompress --help
Usage: tamp decompress [ARGS] [OPTIONS]

Decompress an input file or stream.

— Parameters␣
↪──────────────────────────────────────────────────────────────────────────────
│ INPUT,--input    -i  Input file to decompress. Defaults to stdin.          ␣
↪    │
│ OUTPUT,--output  -o  Output decompressed file. Defaults to stdout.         ␣
↪    │
──────────────────────────────────────────────────────────────────────────────
```

Example usage:

```
tamp decompress enwik8.tamp -o enwik8
echo "hello world" | tamp compress | tamp decompress
```

3.2 Python

The python library can perform one-shot compression, as well as operate on files/streams.

```python
import tamp

# One-shot compression
string = b"I scream, you scream, we all scream for ice cream."
compressed_data = tamp.compress(string)
reconstructed = tamp.decompress(compressed_data)
assert reconstructed == string

# Streaming compression
with tamp.open("output.tamp", "wb") as f:
    for _ in range(10):
        f.write(string)

# Streaming decompression
with tamp.open("output.tamp", "rb") as f:
    reconstructed = f.read()
```

CHAPTER

FOUR

BENCHMARK

In the following section, we compare Tamp against:

- zlib, a python builtin gzip-compatible DEFLATE compression library.
- heatshrink, a data compression library for embedded/real-time systems. Heatshrink has similar goals as Tamp.

All of these are LZ-based compression algorithms, and tests were performed using a 1KB (10 bit) window. Since zlib already uses significantly more memory by default, the lowest memory level (`memLevel=1`) was used in these benchmarks. It should be noted that higher zlib memory levels will having greater compression ratios than Tamp. Currently, there is no micropython-compatible zlib or heatshrink compression implementation, so these numbers are provided simply as a reference.

4.1 Compression Ratio

The following table shows compression algorithm performance over a variety of input data sourced from the Silesia Corpus and Enwik8. This should give a general idea of how these algorithms perform over a variety of input data types.

dataset	raw	tamp	zlib	heatshrink
enwik8	100,000,000	**51,635,633**	56,205,166	56,110,394
build/silesia/dickens	10,192,446	**5,546,761**	6,049,169	6,155,768
build/silesia/mozilla	51,220,480	25,121,385	**25,104,966**	25,435,908
build/silesia/mr	9,970,564	5,027,032	**4,864,734**	5,442,180
build/silesia/nci	33,553,445	8,643,610	**5,765,521**	8,247,487
build/silesia/ooffice	6,152,192	**3,814,938**	4,077,277	3,994,589
build/silesia/osdb	10,085,684	**8,520,835**	8,625,159	8,747,527
build/silesia/reymont	6,627,202	**2,847,981**	2,897,661	2,910,251
build/silesia/samba	21,606,400	9,102,594	**8,862,423**	9,223,827
build/silesia/sao	7,251,944	**6,137,755**	6,506,417	6,400,926
build/silesia/webster	41,458,703	**18,694,172**	20,212,235	19,942,817
build/silesia/x-ray	8,474,240	7,510,606	**7,351,750**	8,059,723
build/silesia/xml	5,345,280	1,681,687	**1,586,985**	1,665,179

Tamp usually out-performs heatshrink, and is generally very competitive with zlib. While trying to be an apples-to-apples comparison, zlib still uses significantly more memory during both compression and decompression (see next section). Tamp accomplishes competitive performance while using around 10x less memory.

4.2 Memory Usage

The following table shows approximately how much memory each algorithm uses during compression and decompression.

	Compression	Decompression
Tamp	(1 << windowBits)	(1 << windowBits)
ZLib	(1 << (windowBits + 2)) + 7KB	(1 << windowBits) + 7KB
Heatshrink	(1 << (windowBits + 1))	(1 << (windowBits + 1))
Deflate (micropython)	(1 << windowBits)	(1 << windowBits)

All libraries have a few dozen bytes of overhead in addition to the primary window buffer, but are implementation-specific and ignored for clarity here. Tamp uses significantly less memory than ZLib, and half the memory of Heatshrink.

4.3 Runtime

As a rough benchmark, here is the performance (in seconds) of these different compression algorithms on the 100MB enwik8 dataset. These tests were performed on an M1 Macbook Air.

	Compression (s)	Decompression (s)
Tamp (Python Reference)	109.5	76.0
Tamp (C)	16.45	0.142
ZLib	0.98	0.98
Heatshrink (with index)	6.22	0.82
Heatshrink (without index)	41.73	0.82

Heatshrink v0.4.1 was used in these benchmarks. When heathshrink uses an index, an additional `(1 << (windowBits + 1))` bytes of memory are used, resulting in 4x more memory-usage than Tamp. Tamp could use a similar indexing to increase compression speed, but has chosen not to to focus on the primary goal of a low-memory compressor.

To give an idea of Tamp's speed on an embedded device, the following table shows compression/decompression in **bytes/second of the first 100KB of enwik8 on a pi pico (rp2040)** at the default 125MHz clock rate. The C benchmark **does not** use a filesystem nor dynamic memory allocation, so it represents the maximum speed Tamp can achieve. In all tests, a 1KB window (10 bit) was used.

	Compression (bytes/s)	Decompression (bytes/s)
Tamp (MicroPython Viper)	4,300	42,000
Tamp (Micropython Native Module)	12,770	644,000
Tamp (C)	28,500	1,042,524
Deflate (micropython builtin)	6,715	146,477

Tamp resulted in a **51637** byte archive, while Micropython's builtin `deflate` resulted in a larger, **59442** byte archive.

4.4 Binary Size

To give an idea on the resulting binary sizes, Tamp and other libraries were compiled for the Pi Pico (`armv6m`). All libraries were compiled with `-O3`. Numbers reported in bytes.

	Compressor	Decompressor	Compressor + Decompressor
Tamp (MicroPython Viper)	4429	4205	7554
Tamp (MicroPython Native)	3232	3047	5505
Tamp (C)	2008	1972	3864
Heatshrink (C)	2956	3876	6832
uzlib (C)	2355	3963	6318

Heatshrink doesn't include a high level API; in an apples-to-apples comparison the Tamp library would be even smaller.

4.5 Acknowledgement

- Thanks @BitsForPeople for the esp32-optimized compressor implementation.

CHAPTER

FIVE

INSTALLATION

Tamp contains 4 implementations:

1. A reference desktop CPython implementation that is optimized for readability (and **not** speed).
2. A Micropython Native Module implementation (fast).
3. A Micropython Viper implementation (not recommended, please use Native Module).
4. A C implementation (with python bindings) for accelerated desktop use and to be used in C projects (very fast).

5.1 Desktop Python

The Tamp library and CLI requires Python >=3.8 and can be installed via:

```
pip install tamp
```

To install directly from github, you can run:

```
python -m pip install git+https://github.com/BrianPugh/tamp.git
```

For development, its recommended to use Poetry:

```
git clone https://github.com/BrianPugh/tamp.git
cd tamp
poetry install
```

5.2 MicroPython

5.2.1 Native Module

Tamp provides pre-compiled *native modules* that are easy to install, are small, and are incredibly fast.

Download the appropriate `.mpy` file from the release page.

- Match the micropython version.
- Match the architecture to the microcontroller (e.g. `armv6m` for a pi pico).

Rename the file to `tamp.mpy` and transfer it to your board. If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = "https://github.com/BrianPugh/tamp/releases/download/v1.4.0/tamp-1.4.0-mpy1.22-
↪armv6m.mpy"
```

5.2.2 Viper

NOT RECOMMENDED, PLEASE USE NATIVE MODULE

For micropython use, there are 3 main files:

1. `tamp/__init__.py` - Always required.
2. `tamp/decompressor_viper.py` - Required for on-device decompression.
3. `tamp/compressor_viper.py` - Required for on-device compression.

For example, if on-device decompression isn't used, then do not include `decompressor_viper.py`. If manually installing, just copy these files to your microcontroller's `/lib/tamp` folder.

If using mip, tamp can be installed by specifying the appropriate `package-*.json` file.

```
mip install github:brianpugh/tamp  # Defaults to package.json: Compressor & Decompressor
mip install github:brianpugh/tamp/package-compressor.json  # Compressor only
mip install github:brianpugh/tamp/package-decompressor.json  # Decompressor only
```

If using Belay, tamp can be installed by adding the following to `pyproject.toml`.

```
[tool.belay.dependencies]
tamp = [
   "https://github.com/BrianPugh/tamp/blob/main/tamp/__init__.py",
   "https://github.com/BrianPugh/tamp/blob/main/tamp/compressor_viper.py",
   "https://github.com/BrianPugh/tamp/blob/main/tamp/decompressor_viper.py",
]
```

5.3 C

Copy the `tamp/_c_src/tamp` folder into your project. For more information, see *C Library*.

CHAPTER

SIX

SPECIFICATION

Tamp uses a single-pass DEFLATE-like algorithm that is optimized for a fast, simple implementation in micropython. Trade-offs were made between code-complexity, speed, memory-usage, and compression-ratio. This document describes the compression algorithm, design choices, and the data format. Tamp outputs a continuous bit stream, where the most-significant-bit (MSb) comes first.

6.1 Stream Header

Tamp has a single header byte described in the following table. The locations are zero index from the beginning of the stream. The bit-location 0 is equivalent to typical MSb position 7 of the first byte.

Bits	Name	Description
[7,6,5]	window	Number of bits, minus 8, used to represent the size of the shifting window. e.g. A 12-bit window is encoded as the number 4, `0b100`. This means the smallest window is 256 bytes, and largest is 32768.
[4,3]	literal_size	Number of bits, minus 5, in a literal. For example, `0b11` represents a standard 8 bit (1 byte) literal.
[2]	custom_dictio	A custom dictionary initialization method was used and must be provided at decompression.
[1]	reserved	Reserved for future use. Must be 0.
[0]	more_head	If `True`, then the next byte in the stream is more header data. Currently always `False`, but allows for future expandability.

6.2 Stream Encoding/Decoding

After the header bytes is the data stream. The datastream is written in bits, so all data is packed tightly with no padding. At a high level, Tamp applies LZSS compression, followed by a fixed, pre-defined Huffman coding for representing the match length.

6.2.1 LZSS

Tamp uses a *slightly* modified LZSS compression algorithm. Modifications are made to make the implementation simpler/faster.

1. Initialize a ring_buffer of size `1 << window` defined in the header. See Dictionary Initialization for initialization details.
2. Starting at the beginning of the plaintext, find the longest match existing in the dictionary. If no pattern (<2 character match) is found, output a literal. If a pattern is detected
   a. literal: `0b1 | literal`. The first bit (1) represents that the following bits represent a literal. The `literal` is `literal_size` bits long.
   b. token: `0b0 | length-huffman-code | offset`. The first bit (`0`) represents that the following bits represent a token. The length of the pattern match is encoded via a pre-defined static Huffman code. Finally, the `offset` is `window` bits long, and points at the offset from the beginning of the dictionary buffer to the pattern. The shortest pattern-length is either going to be 2 or 3 bytes, depending on `window` and `literal` parameters. The shortest pattern-length encoding must be shorter than an equivalent stream of literals. The longest pattern-length will the minimum pattern-length plus 13.

Classically, the `offset` is from the current position in the buffer. Doing so results in the `offset` distribution slightly favoring smaller numbers. Intuitively, it makes sense that patterns are more likely to occur closer to the current text. A proposed idea was to include the `offset` most-significant bit with the length-huffman-code. Unfortuneately, the probability distribution wasn't biased enough to increase compression. For this reason, we just encode the simple offset from the beginning of the dictionary. This is easier to implement and has the potential to execute quicker.

Similarly, attempts to include the `is_literal` flag in the huffman coding did not increase compression. An explicit `is_literal` flag made the code faster and simpler.

6.2.2 Dictionary Initialization

For short messages, having a better initial dictionary can help improve compression ratios. The amount of improvement would be dependent on the type of data being compressed. Given that the contents of raw-binary data could be anything, we chose to focus on improving typical english text. In order to save device space, a whole dictionary is not saved to disk. Instead, we take 16 common characters " x000ei>to<ansnr/." and pseudo-randomly fill up the dictionary with these characters. We use the XorShift32 pseudo-random number generator due to it's good randomness characteristics, and simple implementation.

The chosen seed, 3758097560, was discovered experimentally to give typically good results.

All of this amounts to a few percent compression improvement for short messages.

```
def _xorshift32(seed):
    while True:
        seed ^= (seed << 13) & 0xFFFFFFFF
        seed ^= (seed >> 17) & 0xFFFFFFFF
        seed ^= (seed << 5) & 0xFFFFFFFF
        yield seed


def initialize_dictionary(size):
    chars = b" \x000ei>to<ans\nr/."  # 16 most common chars in dataset

    def _gen_stream(xorshift32):
        for _ in range(size >> 3):
```

(continues on next page)

(continued from previous page)

```
            value = next(xorshift32)
            yield chars[value & 0x0F]
            yield chars[value >> 4 & 0x0F]
            yield chars[value >> 8 & 0x0F]
            yield chars[value >> 12 & 0x0F]
            yield chars[value >> 16 & 0x0F]
            yield chars[value >> 20 & 0x0F]
            yield chars[value >> 24 & 0x0F]
            yield chars[value >> 28 & 0x0F]

    return bytearray(_gen_stream(_xorshift32(3758097560)))
```

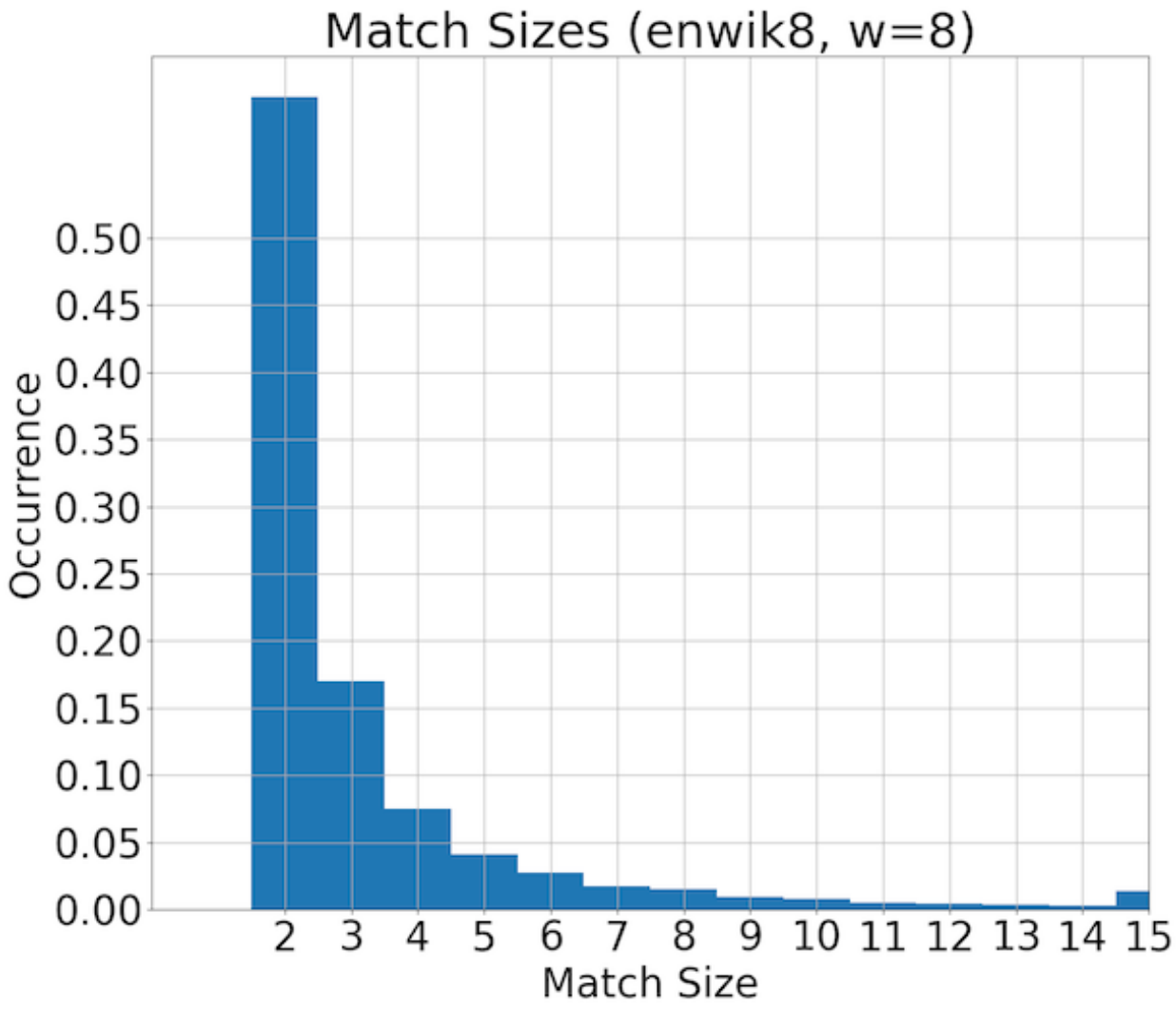
6.2.3 Huffman Coding

Huffman coding encodes high-probability values with less bits, and less-likely values with more bits. In order for huffman coding to work, no encoding is allowed to be a prefix of another encoding. If all values have equal probability, simple binary encoding is more efficient.

The following maps the pattern-size (to be added to the minimum pattern-length) to the bits representing the huffman code.

```
huffman_coding = {
    0: 0b0,
    1: 0b11,
    2: 0b1000,
    3: 0b1011,
    4: 0b10100,
    5: 0b100100,
    6: 0b100110,
    7: 0b101011,
    8: 0b1001011,
    9: 0b1010100,
    10: 0b10010100,
    11: 0b10010101,
    12: 0b10101010,
    13: 0b100111,
    "FLUSH": 0b10101011,
}
```

The match-size probabilities that generated this table were generated over the enwik8 dataset. This huffman coding was chosen such that the longest huffman code is 8 bits long, making it easier to store and index into. The maximum match-size is more likely than the second-highest match-size because all match-sizes greater than the maximum size get down-mapped.



For any given huffman coding schema, a equivalent coding can be obtained by inverting all the bits (reflecting the huffman tree). The single-bit, most common code `0b0` representing a pattern-size 2 is intentionally represented as `0b0` instead of `0b1`. This makes the MSb of all other codes be 1, simplifying the decoding procedure because the number of bits read doesn't strictly have to be recorded.

Flush Symbol

A special FLUSH symbol is encoded as the least likely Huffman code. In many compression algorithms, a `flush()` can only be called at the end of the compression stream, and the compressor cannot be used anymore. In microcontroller applications, the user may want to flush the compressor buffer while still continuing to compress more data. Examples include:

1. Flushing a chunk of logs to disk to prepare if power is removed.
2. Pushing a chunk of collected data to a remote server.

Internally, Tamp uses a 1-byte buffer to store compressed bits until a full byte is available for writing. Invoking the `flush` method can have one of two results:

1. If the buffer **is** empty, no action is performed.

2. If the buffer **is not** empty, then the FLUSH Huffman code is written. No `offset` bits are written following the FLUSH code. The remaining buffer bits are zero-padded and flushed.

On reading, if a FLUSH is read, the reader will discard the remainder of it's 1-byte buffer. In the best-case-scenario (write buffer is empty), a FLUSH symbol will not be emitted. In the worst-case-scenario (1 bit in the write buffer), a FLUSH symbol (9 bits) and the remaining empty 6 bits are flushed. This adds 15 bits of overhead to the output stream.

At the very end of a stream, the FLUSH symbol is unnecessary and **may be omitted** to save an additional one or two bytes.

6.2.4 Miscellaneous

No terminating character is builtin. Tamp relies on external framing (such as from the filesystem) to know when the data stream is complete. The final byte of a stream is zero-padded. The maximum padding is 7 zero bits.

CHAPTER

SEVEN

PYTHON API

class tamp.**Compressor**(*f*, *, *window: int = 10*, *literal: int = 8*, *dictionary: bytearray | None = None*)

Compresses data to a file or stream.

Parameters

- **f** (*Union[str, Path, FileLike]*) -- Path/FileHandle/Stream to write compressed data to.
- **window** (*int*) -- Size of window buffer in bits. Higher values will typically result in higher compression ratios and higher computation cost. A same size buffer is required at decompression time. Valid range: `[8, 15]`. Defaults to `10` (`1024` byte buffer).
- **literal** (*int*) -- Number of used bits in each byte of data. The default `8` bits can store all data. A common other value is `7` for storing ascii characters where the most-significant-bit is always 0. Smaller values result in higher compression ratios for no additional computation cost. Valid range: `[5, 8]`.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At decompression time, the same initialized buffer must be provided. `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

write(*data: bytes | bytearray*) → int

Compress `data` to stream.

Parameters
data (*Union[bytes, bytearray]*) -- Data to be compressed.

Returns
Number of compressed bytes written. May be zero when data is filling up internal buffers.

Return type
int

flush(*write_token: bool = True*) → int

Flushes all internal buffers.

This compresses any data remaining in the input buffer, and flushes any remaining data in the output buffer to disk.

Parameters
write_token (*bool*) -- If appropriate, write a `FLUSH` token. Defaults to `True`.

Returns
Number of compressed bytes flushed to disk.

Return type
int

`close`() → int

Flushes all internal buffers and closes the output file or stream, if tamp opened it.

Returns
Number of compressed bytes flushed to disk.

Return type
int

`__enter__`() → *Compressor*

Use *Compressor* as a context manager.

```
with tamp.Compressor("output.tamp") as f:
    f.write(b"foo")
```

`__exit__`(*exc_type*, *exc_value*, *traceback*)

Calls *close()* on contextmanager exit.

class tamp.**TextCompressor**(*f*, *, *window: int = 10*, *literal: int = 8*, *dictionary: bytearray | None = None*)

Compresses text to a file or stream.

Parameters

- **f** (*Union[str, Path, FileLike]*) -- Path/FileHandle/Stream to write compressed data to.
- **window** (*int*) -- Size of window buffer in bits. Higher values will typically result in higher compression ratios and higher computation cost. A same size buffer is required at decompression time. Valid range: `[8, 15]`. Defaults to `10` (`1024` byte buffer).
- **literal** (*int*) -- Number of used bits in each byte of data. The default `8` bits can store all data. A common other value is `7` for storing ascii characters where the most-significant-bit is always 0. Smaller values result in higher compression ratios for no additional computation cost. Valid range: `[5, 8]`.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At decompression time, the same initialized buffer must be provided. `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

`write`(*data: str*) → int

Compress `data` to stream.

Parameters
data (*Union[bytes, bytearray]*) -- Data to be compressed.

Returns
Number of compressed bytes written. May be zero when data is filling up internal buffers.

Return type
int

`__enter__`() → *Compressor*

Use *Compressor* as a context manager.

```
with tamp.Compressor("output.tamp") as f:
    f.write(b"foo")
```

__exit__(*exc_type*, *exc_value*, *traceback*)

Calls *close()* on contextmanager exit.

close() → int

Flushes all internal buffers and closes the output file or stream, if tamp opened it.

Returns
Number of compressed bytes flushed to disk.

Return type
int

flush(*write_token: bool = True*) → int

Flushes all internal buffers.

This compresses any data remaining in the input buffer, and flushes any remaining data in the output buffer to disk.

Parameters
write_token (*bool*) -- If appropriate, write a `FLUSH` token. Defaults to `True`.

Returns
Number of compressed bytes flushed to disk.

Return type
int

tamp.**compress**(*data: bytes | str*, ***, *window: int = 10*, *literal: int = 8*, *dictionary: bytearray | None = None*) → bytes

Single-call to compress data.

Parameters

- **data** (*Union[str, bytes]*) -- Data to compress.
- **window** (*int*) -- Size of window buffer in bits. Higher values will typically result in higher compression ratios and higher computation cost. A same size buffer is required at decompression time. Valid range: `[8, 15]`. Defaults to `10` (`1024` byte buffer).
- **literal** (*int*) -- Number of used bits in each byte of data. The default `8` bits can store all data. A common other value is `7` for storing ascii characters where the most-significant-bit is always 0. Valid range: `[5, 8]`.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At decompression time, the same initialized buffer must be provided. `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

Returns
Compressed data

Return type
bytes

class tamp.**Decompressor**(*f*, ***, *dictionary: bytearray | None = None*)

Decompresses a file or stream of tamp-compressed data.

Can be used as a context manager to automatically handle file opening and closing:

```
with tamp.Decompressor("compressed.tamp") as f:
    decompressed_data = f.read()
```

Parameters

- **f** (*Union[file, str]*) -- File-like object to read compressed bytes from.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At compression time, the same initialized buffer must be provided. Decompression stream's `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

readinto(*buf: bytearray*) → int

Decompresses data into provided buffer.

Parameters
buf (*bytearray*) -- Buffer to decode data into.

Returns
Number of bytes decompressed into buffer.

Return type
int

read(*size: int = -1*) → bytearray

Decompresses data to bytes.

Parameters
size (*int*) -- Maximum number of bytes to return. If a negative value is provided, all data will be returned. Defaults to `-1`.

Returns
Decompressed data.

Return type
bytearray

close()

Closes the input file or stream, if tamp opened it.

__enter__()

Use *Decompressor* as a context manager.

```
with tamp.Decompressor("output.tamp") as f:
    decompressed_data = f.read()
```

__exit__(*exc_type*, *exc_value*, *traceback*)

Calls *close()* on contextmanager exit.

class tamp.**TextDecompressor**(*f*, *, *dictionary: bytearray | None = None*)

Decompresses a file or stream of tamp-compressed data into text.

Parameters

- **f** (*Union[file, str]*) -- File-like object to read compressed bytes from.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At compression time, the same initialized buffer must be provided. Decompression stream's `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

read(*size: int = -1*) → str

Decompresses data to text.

Parameters
size (*int*) -- Maximum number of bytes to return. If a negative value is provided, all data will be returned. Defaults to `-1`.

Returns
Decompressed text.

Return type
str

__enter__()

Use *Decompressor* as a context manager.

```
with tamp.Decompressor("output.tamp") as f:
    decompressed_data = f.read()
```

__exit__(*exc_type*, *exc_value*, *traceback*)

Calls *close()* on contextmanager exit.

close()

Closes the input file or stream, if tamp opened it.

readinto(*buf: bytearray*) → int

Decompresses data into provided buffer.

Parameters
buf (*bytearray*) -- Buffer to decode data into.

Returns
Number of bytes decompressed into buffer.

Return type
int

tamp.**decompress**(*data: bytes*, *, *dictionary: bytearray | None = None*) → bytearray

Single-call to decompress data.

Parameters

- **data** (*bytes*) -- Tamp-compressed data to decompress.
- **dictionary** (*Optional[bytearray]*) -- Use the given **initialized** buffer inplace. At compression time, the same initialized buffer must be provided. Decompression stream's `window` must agree with the dictionary size. If providing a pre-allocated buffer, but with default initialization, it must first be initialized with *initialize_dictionary()*

Returns
Decompressed data.

Return type
bytearray

tamp.**open**(*f*, *mode='rb'*, ***kwargs*)

Opens a file for compressing/decompressing.

Example usage:

```
with tamp.open("file.tamp", "w") as f:
    # Opens a compressor in text-mode
    f.write("example text")

with tamp.open("file.tamp", "r") as f:
    # Opens a decompressor in text-mode
    assert f.read() == "example text"
```

Parameters

- **f** (*Union[str, Path]*) -- PathLike object to open.
- **mode** (*str*) -- Opening mode. Must be some combination of `{"r", "w", "b"}`.
  - Read-text-mode (`"r"`) will return a *tamp.TextDecompressor*. Read data will be `str`.
  - Read-binary-mode (`"rb"`) will return a *tamp.Decompressor*. Read data will be `bytes`.
  - Write-text-mode (`"w"`) will return a *tamp.TextCompressor*. `str` must be provided to *write()*.
  - Write-binary-mode (`"wb"`) will return a *tamp.Compressor*. `bytes` must be provided to *write()*.
- **kwargs** -- Passed along to class constructor.

Returns
File-like object for compressing/decompressing.

`tamp.`**initialize_dictionary**(*source*, *seed=None*)

Initialize Dictionary.

Parameters
size (*Union[int, bytearray]*) -- If a `bytearray`, will populate it with initial data. If an `int`, will allocate and initialize a bytearray of indicated size.

Returns
Initialized window dictionary.

Return type
bytearray

exception `tamp.`**ExcessBitsError**

Provided data has more bits than expected `literal` bits.

CHAPTER

EIGHT

C LIBRARY

Tamp provides a C library optimized for low-memory-usage, fast runtime, and small binary footprint. This page describes how to use the provided library.

8.1 Overview

To use Tamp in your C project, simply copy the contents of `tamp/_c_src` into your project. The contents are broken down as follows (header files described, but you'll also need the c files):

1. `tamp/common.h` - Common functionality needed by both the compressor and decompressor. Must be included.
2. `tamp/compressor.h` - Functions to compress a data stream.
3. `tamp/decompressor.h` - Functions to decompress a data stream.

All header files are well documented. Please refer to the appropriate header file for precise API usage. This document primarily serves as suggestions on how to use the library, and some of the philosophy behind it.

8.2 Compressor

To include the compressor functionality, include the compressor header:

```
# include "tamp/compressor.h"
```

8.2.1 Initialization

All compression is performed using a `TampCompressor` object. The object must first be initialized with `tamp_compressor_init`. The compressor object, a configuration, and a buffer is provided. Tamp performs no internal allocations, so a buffer must be provided. Tamp is an LZ-based compression schema, and this buffer is commonly called a "window buffer" or "dictionary". The size of the provided buffer must be the same size as described by `conf.window`.

```
static unsigned char *window_buffer[1024];

TampConf conf = {
   /* Describes the size of the decompression buffer in bits.
   A 10-bit window represents a 1024-byte buffer.
   Must be in range [8, 15], representing [256, 32678] byte windows. */
   .window = 10,
```

(continues on next page)

(continued from previous page)

```
    /* Number of bits occupied in each plaintext symbol.
    For example, if ASCII text is being encoded, then we could set this
    value to 7 to slightly improve compression ratios.
    Must be in range [5, 8].
    For general use, 8 (the whole byte) is appropriate. */
    .literal = 8,

    /* To improve compression ratios for very short messages, a custom
    buffer initialization could be used.
    For most use-cases, set this to false.*/
    .use_custom_dictionary = false
};
TampCompressor compressor;
tamp_compressor_init(&compressor, &conf, window_buffer);

// TODO: use the initialized compressor object
```

8.2.2 Compression

Once the `TampCompressor` object is initialized, compression of data can be performed. There's a low-level API to accomplish this, as well as a higher-level one.

The low-level workflow loop is as follows:

1. `tamp_compressor_sink` - Sink in a few bytes of data into the compressor's internal input buffer (16 bytes).
2. `tamp_compressor_poll` - Perform a single compression cycle, compressing up to 15 bytes from the input buffer. Compression is most efficient when the input buffer is full.

The sinking operation is computationally cheap, while the poll compression cycle is much more computationally intensive. Breaking the operation up into these two functions allows `tamp_compressor_poll` to be called at a more opportune time in your program.

To use these 2 functions effectively, loop over calling `tamp_compressor_sink`, then `tamp_compressor_poll`.

```
while(input_size > 0 && output_size > 0){
    {
        // Sink Data into input buffer.
        size_t consumed;
        tamp_compressor_sink(compressor, input, input_size, &consumed);
        input += consumed;
        input_size -= consumed;
    }
    {
        // Perform 1 compression cycle on internal input buffer
        size_t chunk_output_written_size;
        res = tamp_compressor_poll(compressor, output, output_size, &chunk_output_
↪written_size);
        output += chunk_output_written_size;
        output_size -= chunk_output_written_size;
        assert(res == TAMP_OK);
    }
}
```

It is common to compress until an input buffer is exhausted, or an output buffer is full. Tamp provides a higher level function, `tamp_compressor_compress` that does exactly this. Note: you may actually want to use `tamp_compressor_compress_flush`, described in the next section.

Both `tamp_compressor_compress` and `tamp_compressor_compress_flush` have callback-variants: `tamp_compressor_compress_cb` and `tamp_compressor_compress_flush_cb`, respectively. These are the same as their non-callback variants, but they take 2 additional arguments:

- `callback`, a function with signature:

  ```
  int callback(void *user_data, size_t bytes_processed, size_t total_bytes);
  ```

  Where `bytes_processed` are the number of input bytes consumed so far, and `total_bytes` are the number of total bytes provided.

- `void *user_data`, arbitrary data to be passed along to the callback.

The callback can be useful for resetting a watchdog, updating a progress bar, etc.

8.2.3 Flushing

Inside the compressor, there may be up to 16 **bytes** of uncompressed data in the input buffer, and 31 **bits** in an output buffer. This means that the compressed output lags behind the input data stream.

For example, if we compress the 44-long non-null-terminated string `"The quick brown fox jumped over the lazy dog"`, the compressor will produce a 32-long data stream, that decompresses to `"The quick brown fox jumped ov"`. The remaining `"er the lazy dog"` is still in the compressor's internal buffers.

To flush the remaining data, use `tamp_compressor_flush` that performs the following actions:

1. Repeatedly call `tamp_compressor_poll` until the 16-byte internal input buffer is empty.
2. Flush the output buffer. If `write_token=true`, then the special `FLUSH` token will be appended if padding was required.

```
tamp_res res;
output_buffer = bytes[100];
size_t output_written;  // Stores the resulting number of bytes written to output_buffer.

res = tamp_compressor_flush(&compressor, output_buffer, sizeof(output_buffer), &output_
↪written, true);
assert(res == TAMP_OK);
```

The special `FLUSH` token allows for the compressor to continue being used, but adds 0~2 bytes of overhead.

1. If intending to continue using the compressor object, then `write_token` should be true.
2. If flushing the compressor to finalize a stream, then setting `write_token` to false will save 0~2 bytes. Setting `write_token` to true will have no impact aside from the extra 0~2 byte overhead.

`tamp_compressor_compress_and_flush` is just like `tamp_compressor_compress`, with the addition that the internal buffers are flushed at the end of the call.

8.2.4 Summary

```c
unsigned char *window_buffer[1024];
const unsigned char input_string[44] = "The quick brown fox jumped over the lazy dog";
unsigned char output_buffer[64];

TampConf conf = {.window=10, .literal=8};
TampCompressor compressor;
tamp_compressor_init(&compressor, &conf, window_buffer);

size_t input_consumed_size, output_written_size;
tamp_compressor_compress_and_flush(
     &compressor,
     output_buffer, sizeof(output_buffer), &output_written_size,
     input_string, sizeof(input_string), &input_consumed_size,
     false  // Don't write flush token
);

// Compressed data is now in output_buffer
printf("Compressed size: %d\n", output_written_size);
```

8.3 Decompressor

The decompressor API is much simpler than the compressor API. To include the decompressor functionality, include the decompressor header:

```c
# include "tamp/decompressor.h"
```

8.3.1 Initialization

All decompression is performed using a `TampDecompressor` object. Like `TampCompressor`, this object needs to be configured with a `TampConf` object. Typically, this configuration comes from the Tamp header at the beginning of the compressed data. Use `tamp_decompressor_read_header` to read the header into a `TampConf`:

```c
const unsigned char compressed_data[64];  // Imagine this contains tamp-compressed data.
sizez_t compressed_data_size = 64;
tamp_res res;
TampConf conf;
size_t compressed_consumed_size;

// This will populate conf.
res = tamp_decompressor_read_header(
    &conf,
    compressed_data, compressed_data_size, &compressed_consumed_size
);
assert(res == TAMP_OK);

compressed_data += compressed_consumed_size;
compressed_data_size -= compressed_consumed_size;
```

(continues on next page)

(continued from previous page)

```
// TODO: actual decompression.
```

Explicitly reading the header is useful if the window-buffer needs to be dynamically allocated. The window-buffer size can be calculated as `(1 << conf.window)`. If a static window buffer is used, then `tamp_decompressor_read_header` doesn't need to be explicitly called. `tamp_decompressor_init` initializes the actual decompressor object, using an optionally supplied `TampConf`. If no `TampConf` is provided, then it will be automatically initialized on first `tamp_decompressor_decompress` call from input header data.

```
TampDecompressor decompressor;
unsigned char window_buffer[1024];
tamp_res res;

// Since no TampConf is provided, the header will automatically be parsed
// in the first tamp_decompressor_decompress call.
res = tamp_decompressor_init(&decompressor, NULL, window_buffer);

assert(res == TAMP_OK);
```

8.3.2 Decompression

Data decompression is straight forward:

```
const unsigned char input_data[64]; // Hypothetical input compressed data.
size_t input_consumed_size;

unsigned char output_data[64];  // output decompressed data
size_t output_written_size;

res = tamp_decompressor_decompress(
    &decompressor,
    output_data, sizeof(output_data), &output_written_size,
    input_data, sizeof(input_data), &input_consumed_size
);
// res could be:
//    TAMP_INPUT_EXHAUSTED - All data in input buffer has been consumed.
//    TAMP_OUTPUT_FULL - Output buffer is full.
// In all situations, output_written_size and input_consumed_size is updated.
```

`tamp_decompressor_decompress` has a callback-variant: `tamp_decompressor_decompress_cb`. These are the same as their non-callback variants, but they take 2 additional arguments:

- `callback`, a function with signature:

  ```
  int callback(void *user_data, size_t bytes_processed, size_t total_bytes);
  ```

  Where `bytes_processed` are the number of input bytes consumed so far, and `total_bytes` are the number of total input bytes provided.

- `void *user_data`, arbitrary data to be passed along to the callback.

The callback can be useful for resetting a watchdog, updating a progress bar, etc. Compred to compression, decompression is very very fast; it is unlikely that the decompression callback feature provides significant value.

INDEX

Symbols

B

C

D

E

F

I

R

T

W