# SHA-256 Hashing Function Project
# ECRYP 23Z

**Authors**

Marcel Piotrowski - 303108
Aleksander Szweryn - 310723

January 3, 2024

# Contents

# 1 Theoretical introduction

## 1.1 Hashing

Before we start, it is important to distinguish between hashing and other types of functions in terms of cryptography. Hashing is a process of producing a fixed-sized string of characters from given input data. It is common to produce hexadecimal digest of an input string. There are two most important characteristics of hashing:

- Irreversibility - It should be computationally infeasible to reverse the process and obtain the original input from the hash value.
- Determinism - The same hashing function for the same input value should always produce the same result.

## 1.2 SHA-256

One of most popular example of hashing functions is SHA-256 (Secure Hash Algorithm 256-bit), which is a member of SHA-2 family of cryptographic hash functions. It was designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) as a part of the Digital Signature Algorithm (DSA) in 2001. SHA-256 is widely used for various cryptographic applications and is considered to be secure. The "256" name part indicates, that the output is of 256-bit length.

Here's a high-level overview of the SHA-256 hashing process:

### 1.2.1 Message Padding

The input message is padded to ensure its length is a multiple of 512 bits. Padding includes the original message length in bits.

### 1.2.2 Initial Hash Values

SHA-256 uses eight initial hash values (H0 through H7), which are derived from the fractional parts of the square roots of the first eight prime numbers.

### 1.2.3 Breaking into Blocks

The padded message is divided into blocks of 512 bits. The 512-bit block is further divided into sixteen 32-bit words, forming the message schedule array (W[0] to W[15]).

### 1.2.4 Processing Each Block

The compression function is applied to each block. The function uses a series of logical functions (AND, OR, XOR), bitwise operations (shifts, rotates), and modular additions.

The compression function processes the message schedule and the current hash values to produce new hash values.

### 1.2.5 Iterations

The compression function is iteratively applied for a total of 64 rounds. Different logical functions and constants are used in each round.

### 1.2.6 Final Hash Value

After processing all blocks, the final hash value is the concatenation of the eight 32-bit hash values (H0 to H7). The cryptographic strength of SHA-256 comes from its properties, such as being collision-resistant (extremely unlikely for two different inputs to produce the same hash) and resistant to preimage attacks (given a hash value, it's computationally infeasible to determine the original input).

# 2 Functional description of the application

## 2.1 Usage

The designed usage of our application project is a Python3 command line interface (CLI) tool. The program requires installed Python environment, and additionally *pytest* module for running implemented test cases. Usually, preparation before using the application involves installing Python virtual environment (*venv*), enabling it, then downloading requirements using Python installation manager (*pip*). Example using Windows 11 command line below:

```
python -m venv env
env\Scripts\activate
pip install -r requirements.txt
```

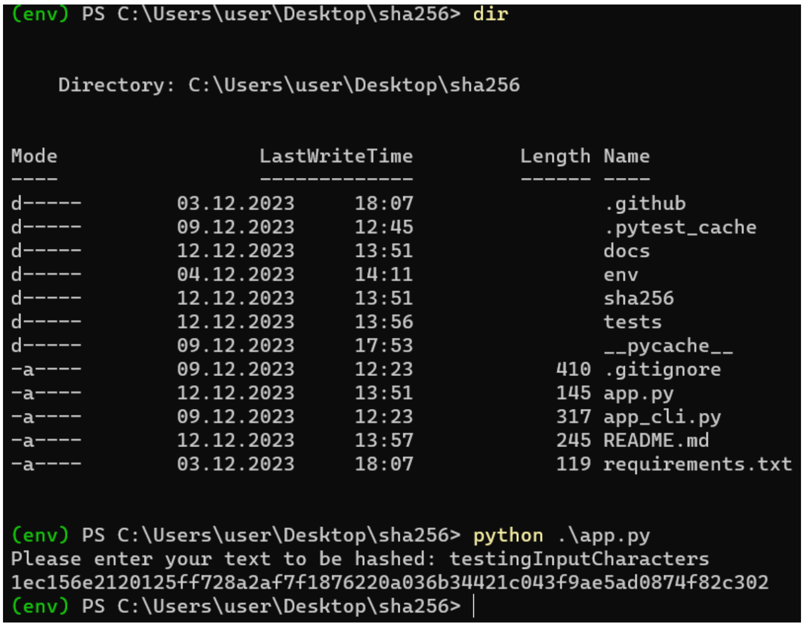
Then to use the CLI tool:

```
python app_cli.py --help
python app_cli.py textToBeHashed
```

It is also possible that by using the CLI tool, `app_cli.py`, the input message can be read from a file. The result can be also specifed as a file as well. For details, look into the help section of the tool.

There is also a script version of the application, where user runs the script and is prompted to enter input text to be hashed:

```
python app.py
```



```
(env) PS C:\Users\user\Desktop\sha256> dir


    Directory: C:\Users\user\Desktop\sha256


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        03.12.2023     18:07                .github
d-----        09.12.2023     12:45                .pytest_cache
d-----        12.12.2023     13:51                docs
d-----        04.12.2023     14:11                env
d-----        12.12.2023     13:51                sha256
d-----        12.12.2023     13:56                tests
d-----        09.12.2023     17:53                __pycache__
-a----        09.12.2023     12:23            410 .gitignore
-a----        12.12.2023     13:51            145 app.py
-a----        09.12.2023     12:23            317 app_cli.py
-a----        12.12.2023     13:57            245 README.md
-a----        03.12.2023     18:07            119 requirements.txt


(env) PS C:\Users\user\Desktop\sha256> python .\app.py
Please enter your text to be hashed: testingInputCharacters
1ec156e2120125ff728a2af7f1876220a036b34421c043f9ae5ad0874f82c302
(env) PS C:\Users\user\Desktop\sha256> |
```

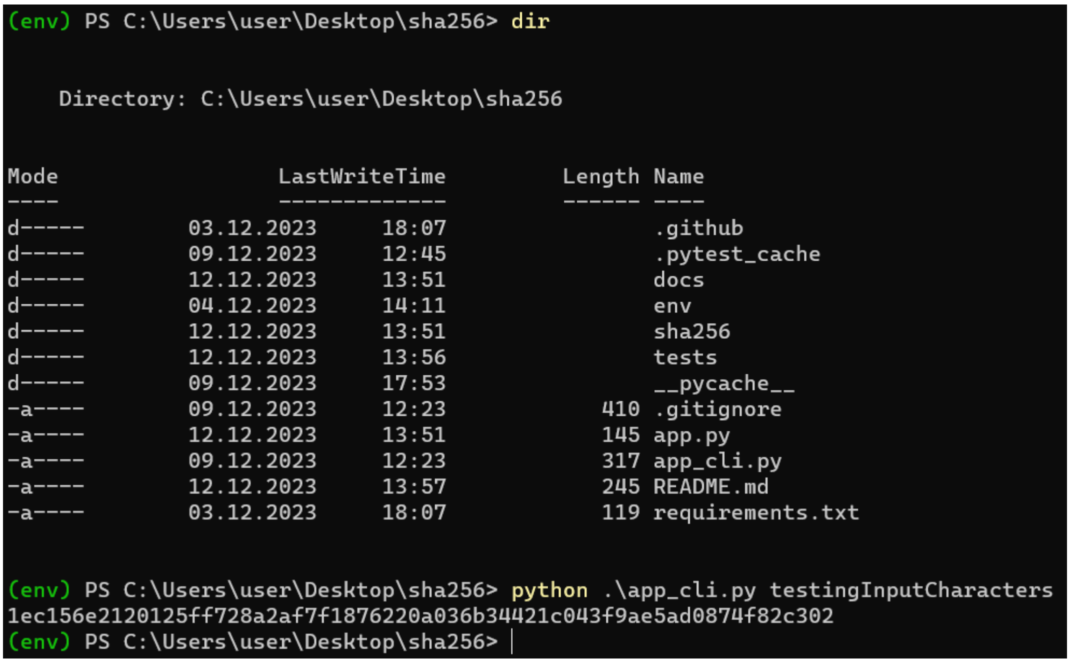
Figure 1: The script version of the application usage with a convenient prompt.

## 2.2 Input

Expected input of the hashing function is a string which theoretically should be of any length.

## 2.3 Output

The output og the hashing function will be given in a hexadecimal string representation of the digest. It can be printed in stdin of the program or saved as to a text file described while passing the "*output_file*" argument.



```
(env) PS C:\Users\user\Desktop\sha256> dir


    Directory: C:\Users\user\Desktop\sha256


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        03.12.2023     18:07                .github
d-----        09.12.2023     12:45                .pytest_cache
d-----        12.12.2023     13:51                docs
d-----        04.12.2023     14:11                env
d-----        12.12.2023     13:51                sha256
d-----        12.12.2023     13:56                tests
d-----        09.12.2023     17:53                __pycache__
-a----        09.12.2023     12:23            410 .gitignore
-a----        12.12.2023     13:51            145 app.py
-a----        09.12.2023     12:23            317 app_cli.py
-a----        12.12.2023     13:57            245 README.md
-a----        03.12.2023     18:07            119 requirements.txt


(env) PS C:\Users\user\Desktop\sha256> python .\app_cli.py testingInputCharacters
1ec156e2120125ff728a2af7f1876220a036b34421c043f9ae5ad0874f82c302
(env) PS C:\Users\user\Desktop\sha256>
```

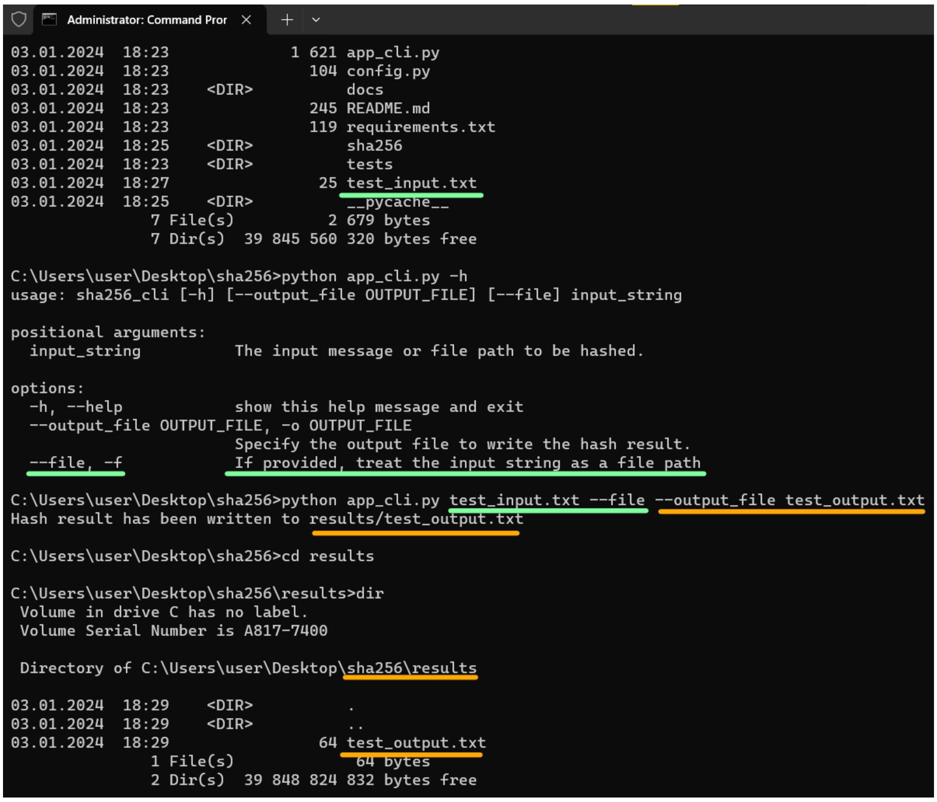
Figure 2: The CLI version of the application.

## 2.4 File I/O mode

To make our CLI application useful e.g. for developers we have prepared file I/O operations. Usage is described when the Python script is called with "-h" or "–help" flag.

### 2.4.1 Output file path

If needed, users can define a file path for the hashing function output which will be saved by default to the relative "results" directory. For this purpose, call the Python script wih "-o" or "–output_file" parameter after which directory name should be described.

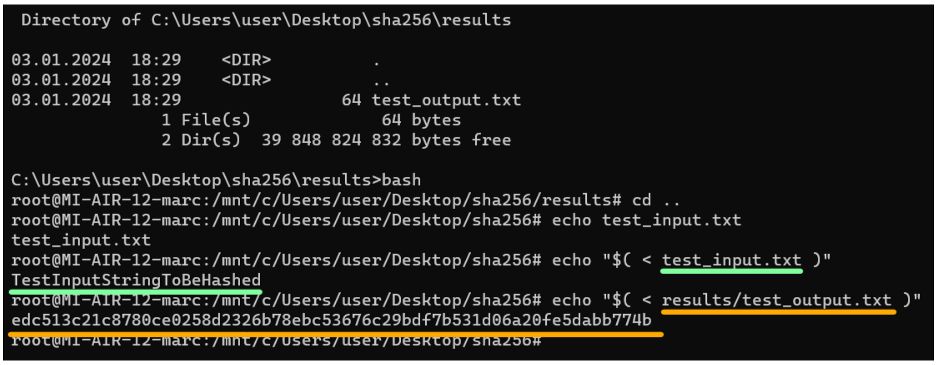


```
03.01.2024  18:23             1 621 app_cli.py
03.01.2024  18:23               104 config.py
03.01.2024  18:23    <DIR>          docs
03.01.2024  18:23               245 README.md
03.01.2024  18:23               119 requirements.txt
03.01.2024  18:25    <DIR>          sha256
03.01.2024  18:23    <DIR>          tests
03.01.2024  18:27                25 test_input.txt
03.01.2024  18:25    <DIR>          __pycache__
               7 File(s)          2 679 bytes
               7 Dir(s)  39 845 560 320 bytes free

C:\Users\user\Desktop\sha256>python app_cli.py -h
usage: sha256_cli [-h] [--output_file OUTPUT_FILE] [--file] input_string

positional arguments:
  input_string          The input message or file path to be hashed.

options:
  -h, --help            show this help message and exit
  --output_file OUTPUT_FILE, -o OUTPUT_FILE
                        Specify the output file to write the hash result.
  --file, -f            If provided, treat the input string as a file path

C:\Users\user\Desktop\sha256>python app_cli.py test_input.txt --file --output_file test_output.txt
Hash result has been written to results/test_output.txt

C:\Users\user\Desktop\sha256>cd results

C:\Users\user\Desktop\sha256\results>dir
 Volume in drive C has no label.
 Volume Serial Number is A817-7400

 Directory of C:\Users\user\Desktop\sha256\results

03.01.2024  18:29    <DIR>          .
03.01.2024  18:29    <DIR>          ..
03.01.2024  18:29                64 test_output.txt
               1 File(s)             64 bytes
               2 Dir(s)  39 848 824 832 bytes free
```

Figure 3: Example usage of "-f" and "-o" aliases when using the command line interface.

### 2.4.2 Input file path

In case of large inputs it is recommended to write the text input to be hashed into the file (for example .txt file). Afterwards our Python script can be called with "-f" or "–file" flag which results in changing how the input string is processed, namely our program treats it as a file path to the file which contents are to be hashed.



```
 Directory of C:\Users\user\Desktop\sha256\results

03.01.2024  18:29    <DIR>          .
03.01.2024  18:29    <DIR>          ..
03.01.2024  18:29                64 test_output.txt
               1 File(s)             64 bytes
               2 Dir(s)  39 848 824 832 bytes free

C:\Users\user\Desktop\sha256\results>bash
root@MI-AIR-12-marc:/mnt/c/Users/user/Desktop/sha256/results# cd ..
root@MI-AIR-12-marc:/mnt/c/Users/user/Desktop/sha256# echo test_input.txt
test_input.txt
root@MI-AIR-12-marc:/mnt/c/Users/user/Desktop/sha256# echo "$( < test_input.txt )"
TestInputStringToBeHashed
root@MI-AIR-12-marc:/mnt/c/Users/user/Desktop/sha256# echo "$( < results/test_output.txt )"
edc513c21c8780ce0258d2326b78ebc53676c29bdf7b531d06a20fe5dabb774b
root@MI-AIR-12-marc:/mnt/c/Users/user/Desktop/sha256#
```

Figure 4: Hashing results of the previous example usage.

# 3 Description of designed code structure

SHA-256 Implementation

In this chapter, we present a comprehensive overview of the SHA-256 implementation, including detailed explanations of each function and the theoretical concepts behind them.

## 3.1 Constants

- `K`: A list of round constants used in the SHA-256 algorithm. These constants are derived from the fractional parts of the cube roots of the first 64 prime numbers.
- `INIT_HASH`: Initial hash values for the SHA-256 algorithm. These values are obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers.
- `MAX_32`: Maximum 32-bit integer value. It is used to perform bitwise AND operations to ensure that hash values remain within the bounds of 32 bits.

## 3.2 Functions

### 3.2.1 `padding(mlen: int) -> bytes`

The function generates padding for SHA-256, aligning the input message length to ensure correct processing by the hash algorithm. For this purpose a Message Digest Index is calculated, which points to the last byte withing the currently processed message block. It is acquired by calculating the reminder of division of the message length `mlen` by 64. After that we address two cases, to obtain a proper length of the padding `padlength`. The padding returned at the end of the function follows the Merkle–Damgård construction, where the message is padded with a single '1' bit followed by zeros and the original message length. This ensures that the final message length is a multiple of the 64-byte block size.

### 3.2.2 `process(c: list, hash: list)`

The `process` function implements the core compression function of SHA-256. It performs bitwise operations, logical functions, and additions on a 512-bit data block `c` and the current hash state `hash`. This function iterates through 64 rounds, applying various operations such as Sigma functions and bitwise rotations, which were implemented by us, as specified by the NIST documentation. Finally, at the end of each loop revolution, current hash state is being updated, and prepared for the next steps.

### 3.2.3 `update(m: bytes, mlen: int, buf: bytes, hash: list) -> Tuple[int, bytes]`

The `update` function processes the input message `m` and updates the hash state `hash`. It manages the total message length and handles the concatenation of partial blocks. The function iterates through the message, processing full 64-byte blocks using the `process` function, and updates the buffer for any remaining partial block. First, the safety measure for the empty input case has been addressed. Later, we update the message length and concatenates the current buffer with the input message, forming a new message. Having that, we can use our already designed `process` function to process all full 64-byte blocks. Then, if any partial blocks are left, they are stored in a buffer. At the end the function return both the updated message length as well as the buffer state after its execution.

### 3.2.4 `digest(mlen: int, buf: bytes, hash: list) -> bytes`

Finalization of the hash computation by ensuring that the entire message is processed. It uses the `update` function to include any remaining partial blocks and converts the hash state into a byte sequence, producing the final hash digest. For the update function to work properly. We also need to use `padding` to ensure the right format. After the computation is complete, part by part, the outcome is converted from bit form into a 4-byte, big-endian representation, which is then concentraded by the join function and the final hash digest is returned.

### 3.2.5 `hex(mlen: int, buf: bytes, hash: list) -> str`

The `hex` function converts the byte values of the hash digest obtained from the `digest` function into a hexadecimal string. This is used simply for the presentation purposes, as the hexadecimal representation of the hashed message is oftenly used for hash presentation.

### 3.2.6 `hash(inp: str) -> str`

Entry point for our SHA-256 hashing simulation. It initializes all the previously discussed functions to conduct the hashing process and return the final result. All required variables are initialized, and then the `hex` function is initialized to obtain the result.

## 3.3 Program flowchart

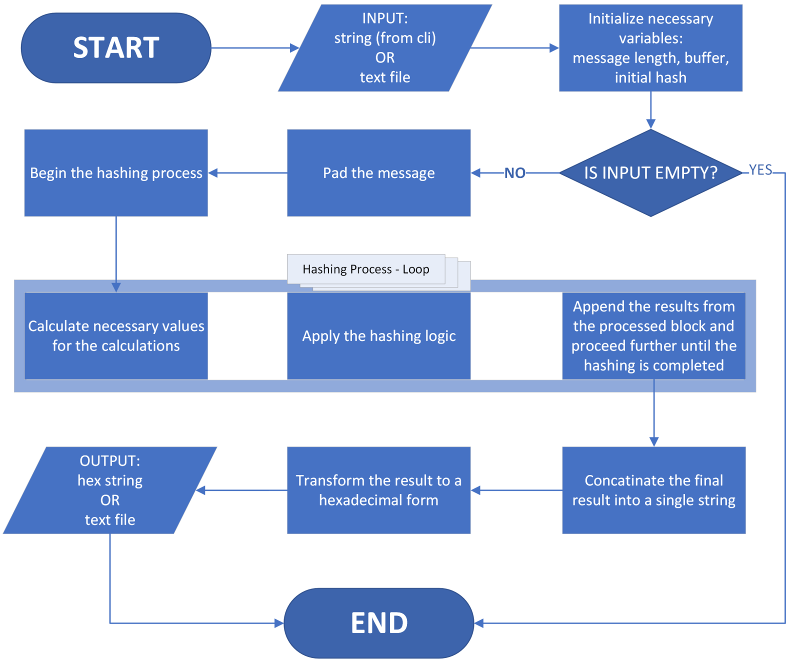

START
INPUT:
string (from cli)
OR
text file
Initialize necessary variables:
message length, buffer, initial hash
IS INPUT EMPTY?
NO
YES
Pad the message
Begin the hashing process
Hashing Process - Loop
Calculate necessary values for the calculations
Apply the hashing logic
Append the results from the processed block and proceed further until the hashing is completed
Concatinate the final result into a single string
Transform the result to a hexadecimal form
OUTPUT:
hex string
OR
text file
END

Figure 5: Simple diagram which describes the code flow in one picture.

# 4 Tests

## 4.1 SHA-256

Our tests are conducted with usage of *pytest* and *hashlib* Python modules. The first one is for convenience of testing the source code, the second gives us some reference values of the hexadecimal SHA-256 digest to verify correctness of our algorithm implementation. All our test cases were successful and their parameters are provided below:

| **INPUT** | **EXPECTED AND TESTED OUTPUT** |
|---|---|
| "" (empty string) | e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 |
| " " (whitespace) | 36a9e7f1c95b82ffb99743e0c5c4ce95d83c9a430aac59f84ef3cbfab6145068 |
| "i" | de7d1b721a1e0632b7cf04edf5032c8ecffa9f9a08492152b926f1a5a7e765d7 |
| "Hello, World!" | dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f |
| "123456" | 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92 |
| "ECRYP CRYPT testing" | ad46a3c190178b6c2bf56a1a8e884d2153ba9e534a2d90aca155ab8a2f6a2307 |
| "Hash Function Test" | 21bda56c22ceaac62744da9c1ac512a8615e300937af3f289df83f4da29f9d93 |
| "python" | 11a4a60b518bf24989d481468076e5d5982884626aed9faeb35b8576fcd223e1 |
| "987654321" | 8a9bcf1e51e812d0af8465a8dbcc9f741064bf0af3b3d08e6b0246437c19f7fb |
| lowercase chars | 77d721c817f9d216c1fb783bcad9cdc20aaa2427402683f1f75dd6dfbe657470 |
| "[" character 32600 times | 14e6ad0b36802e815a03efe9831bc7b7c90096bf06a18b1051c1072ab5fe9c8b |

## 4.2 Operators

We have also prepared test cases for additional bit wise operators that we had to implement in Python. Those were also successful and provided below.

### 4.2.1 ROTATE RIGHT

| **INPUT** | **EXPECTED AND TESTED OUTPUT** |
|---|---|
| 0x00000010 rotate by 4 | 0x00000001 |
| 0x00000001 rotate by 4 | 0x10000000 |
| 0x000000F0 rotate by 4 | 0x0000000F |

### 4.2.2 CH - (a AND b) XOR (NOT a AND c)

| a | b | c | RESULT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

### 4.2.3 MAJ - (a AND b) XOR (a AND c) XOR (b AND c)

| a | b | c | RESULT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

### 4.2.4 File I/O

Additionally, we have also implemented basic testing for optional file input/output operations which check if file is properly created in the results folder.

# 5 List of sources

- National Institute of Standards and Technology. "Secure Hash Standard (SHS)." CSRC, NIST, 4 Aug. 2015, csrc.nist.gov/pubs/fips/180-4/upd1/final
- Real Python website article on Python bitwise operators - https://realpython.com/python-bitwise-operators/
- PyPy GitLab, one of the Python interpreters repository, for SHA-256 implementation reference (file: pypy/lib_pypy/_sha256.py) - https://foss.heptapod.net/pypy/pypy