#### Hashing algorithms

There are many types of hashing algorithms, like cryptography, checksums, data structures, similarity detection, compression and encoding, analytics, monitoring, load balancing, etc.

# I'm going to talk about the following segregation:

- 1. Cryptography hash functions
- 2. Password hashing algorithms
- 3. Checksums hashing
- 4. Data Structures?
- 5. Load balancing?
- 6. Similarity Detection?

## Cryptography hash functions (CHF) PREREQUISITES

- Transforms input data into a fixed-length sequence of characters
- designed to be fast.
- deterministic—the hash function consistently computes the same hash for the same input.
- one-way—the algorithm is made irreversible in the sense that it is computationally impossible to recover the original input from its hash value.
- collision —The function minimizes the chance that two distinct inputs will produce identical hash values

### Cryptography hash functions (CHF) MOST USED

- 1. MD5
- 2. SHA-1
- 3. SHA-2
- 4. SHA-3
- 5. BLAKE2
- 6. BLAKE3

#### Cryptography hash functions (CHF)

MD5 (MESSAGE DIGEST ALGORITHM 5)

```
mds.py > ...
    Run Cell|Run Below|Debug Cell

# %%

import hashlib

def get_md5_of_string(input_string):
    md5_hash = hashlib.md5(input_string.encode('utf-8')).hexdigest()
    return md5_hash

string_to_hash = "Anderson"
    md5_result = get_md5_of_string(string_to_hash)
    print(f"hash of \"{string_to_hash}\": {md5_result}")
    Run Cell|Run Above|Debug Cell

# %%
```

```
Connected to .venv (Python 3.12.3)

✓ import hashlib ···

··· hash of "Anderson": b32b1b822dd59451b17b08f97fdfe81e

✓ import hashlib ···

··· hash of "Anderson": b32b1b822dd59451b17b08f97fdfe81e

✓ import hashlib ···

··· hash of "Anderson": b32b1b822dd59451b17b08f97fdfe81e
```

### Cryptography hash functions (CHF) MD5 COLLISION

```
Run Cell Run Above | Debug Cell

# %%

## simulating a md5 colission

import hashlib

# Source: https://www.johndcook.com/blog/2024/03/20/md5-hash-collision/

string1 = b"TEXTCOLLBYfGiJUETHQ4hAcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"

string2 = b"TEXTCOLLBYfGiJUETHQ4hEcKSMd5zYpgqf1YRDhkmxHkhPWptrkoyz28wnI9V0aHeAuaKnak"

if(string1 != string2):
    print("Input strings are different\n")

hash1 = hashlib.md5(string1).hexdigest()
hash2 = hashlib.md5(string2).hexdigest()

print(f"MD5 Hash 1: {hashl}\n")
print(f"MD5 Hash 2: {hash2}")

if hash1 == hash2:
    print("\nMD5 Collision Detected")
```



# Cryptography hash functions (CHF) SHA-1 (SECURE HASH ALGORITHM 2)

```
sha-1.py > ...
    Run Cell|Run Below|Debug Cell

# %%

## creating a sha-1

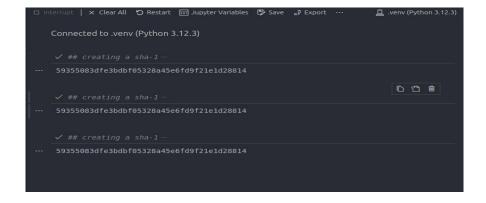
import hashlib

string_to_hash = "Anderson"

hash = hashlib.shal(string_to_hash.encode()).hexdigest()

print(hash)
    Run Cell|Run Above|Debug Cell

# %%
```



#### Cryptography hash functions (CHF) SHA-1 COLLISION

```
Run Cell | Run Above | Debug Cell # % 
import hashlib

def check_if_files_is_different():
    with open("./files/shattered-1.pdf", "rb") as f1, open("./files/shattered-2.pdf", "rb") as f2:
    if f1.read()! = f2.read():
        print("The files are different\n")

def shal_hash(filename):
    with open(filename, 'rb') as f:
    return hashlib.shal(f.read()).hexdigest()

check_if_files_is_different()
    hash1 = shal_hash("./files/shattered-1.pdf")
    hash2 = shal_hash("./files/shattered-2.pdf")

print("hash1: ", hash1)
    print("hash2: ", hash2)

if hash1 == hash2:
    print("\nCollision detected")
```

```
Connected to .venv (Python 3.12.3)

✓ import hashlib —

The files are different

hashl: 38762cf7f55934b34d179ae6a4c80cadccbb7f0a
hash2: 38762cf7f55934b34d179ae6a4c80cadccbb7f0a
Collision detected
```

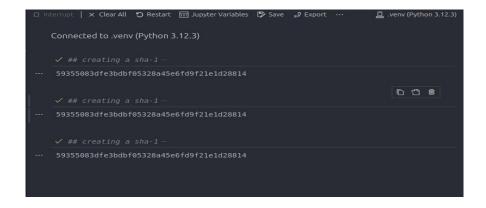
# Cryptography hash functions (CHF) SHA-2 (SECURE HASH ALGORITHM 2)

```
sha-1.py > ...
    Run Cell|Run Below|Debug Cell

# %%
## creating a sha-1

import hashlib
string_to_hash = "Anderson"
hash = hashlib.sha1(string_to_hash.encode()).hexdigest()
print(hash)
Run Cell|Run Above|Debug Cell

# %%
```



### Cryptography hash functions (CHF)

```
blake2.py > ...

Run Cell | Run Below | Debug Cell | You, 3 seconds ago | 1 author (You)

# %%

## Can produce a hash digest of up to 64 bytes (512 bits).

import hashlib

string_to_hash = b"Anderson"

blake_object = hashlib.blake2b(digest_size=64)

blake_object.update(string_to_hash)

hash_result = blake_object.hexdigest()

print(f"Hash: {hash_result}")

Fun Cell | Run Above | Debug Cell
```

### Cryptography hash functions (CHF)

```
Run Cell | Run Above | Debug Cell

9  # %%

10  # Keyed hashing

11  import blake3

12

13  my_key = b"my_key"

14  keyed_hash = blake3.blake3(key=my_key)

15  keyed_hash.update(b"Sensitive information.")

16  print(f"Keyed Hash: {keyed_hash.hexdigest()}")

Run Cell | Run Above | Debug Cell
```

```
import blake3

extended_output = blake3.blake3(b"input_for_xof").digest(length=64)
print(f"Extended_output: {extended_output.hex()}")

seeked_output = blake3.blake3(b"input_for_xof").digest(length=10, seek=10)

print(f"Seeked_Output (bytes 10-19): {seeked_output.hex()}")

Run Cell Run Above [Debug Cell

* %
import blake3

large_input = bytearray(10 * 1024 * 1024) # IC MB of data

large_input = bytearray(10 * 1024 * 1024) # IC MB of data

hash_auto_threads = blake3.blake3(large_input, max_threads=2).hexdigest()
print(f"Hash: {hash_auto_threads}")
```

#### Cryptographic Hash Algorithms TRADEOFFS

| CRITERION        | MD5        | SHA-1      | SHA-2      | SHA-3     | BLAKE2    | BLAKE3         |
|------------------|------------|------------|------------|-----------|-----------|----------------|
| Security         | Broken     | Broken     | Secure     | Secure    | Secure    | Very Secure    |
| Collision        | Weak       | Weak       | Strong     | Strong    | Strong    | Very Strong    |
| Speed            | Fast       | Medium     | Medium     | Slower    | Very Fast | Extremely Fast |
| Length Extension | Vulnerable | Vulnerable | Vulnerable | Resistant | Resistant | Resistant      |

## Password hashing algorithms PREREQUISITES

- one-way process
- deterministic nature
- resistance to collisions
- computational cost
- salting—Salting is a technique where a random value (the salt) is added to the password before hashing.

## Password hashing algorithms

- 1. Bcrypt
- 2. Argon2
- 3. PBKDF2

## Password hashing algorithms BCRYPT

```
assword hashing algorithms > 🤚 bcrypt .py > ...
     print(f"Hashed password: {hashed password.decode('utf-8')}")
     entered byte password = entered password.encode('utf-8')
     print(f"\nTrying to authenticate with password: {entered password}")
         print("Password match! User authenticated.\n")
     print(f"Trying to authenticate with incorrect password: {incorrect password}")
         print("Password match! (This should not happen for incorrect password)")
         print("Incorrect password. Authentication failed (as expected).")
```

## Password hashing algorithms ARGON2

```
password_hashing_algorithms > 🥐 argon2_.py > ...
      import argon2
      salt = b'some salt'
      password = b'password'
      argon2 hasher = argon2.PasswordHasher(
          salt len=16 # defaults to 16
      hashed password = argon2 hasher.hash(password.decode('utf-8'))
      print("Argon2 hash", hashed password)
      password is right = argon2 hasher.verify(hashed password, "password")
      print("Password is right ", password is right)
          argon2 hasher.verify(hashed password, "wrong password")
      except argon2.exceptions.VerifyMismatchError:
          print("Password is wrong")
```

# Password hashing algorithms PBKDF2 (PASSWORD-BASED KEY DERIVATION FUNCTION 2)

```
password_hashing_algorithms > 🥐 argon2_.py > ...
      import argon2
      salt = b'some salt'
      password = b'password'
      argon2 hasher = argon2.PasswordHasher(
          salt len=16 # defaults to 16
      hashed password = argon2 hasher.hash(password.decode('utf-8'))
      print("Argon2 hash", hashed password)
      password is right = argon2 hasher.verify(hashed password, "password")
      print("Password is right ", password is right)
          argon2 hasher.verify(hashed password, "wrong password")
      except argon2.exceptions.VerifyMismatchError:
          print("Password is wrong")
```

#### Password hashing algorithms TRADEOFFS

| CRITERIA                  | PBKDF2             | ARGON2                          | BCRYPT                   |
|---------------------------|--------------------|---------------------------------|--------------------------|
| Resistance to Brute Force | Medium (CPU-bound) | High (CPU + memory-bound)       | Good (CPU-bound)         |
| Resistance to GPU/ASIC    | Low to Medium      | High                            | Medium                   |
| Configurable Parameters   | Iterations         | Iterations, memory, parallelism | Cost factor (log rounds) |
| Speed                     | Fast               | Slow (intentionally)            | Reasonably fast          |
| Memory Usage              | Low                | High (configurable)             | Very low                 |

# Checksum hashing algorithms PREREQUISITES

- fixed-size output
- one-way function
- sensitivity to input changes
- collision resistance
- deterministic

## Checksum hashing algorithms

- 1. CRC32
- 2. Adler-32
- 3. MD5
- 4. SHA-1 (variants)

# Checksum hashing algorithms CRC32 (CYCLIC REDUNDANCY CHECK 32)

```
password_hashing_algorithms > 💨 argon2_.py > ...
      import argon2
      salt = b'some salt'
      password = b'password'
      argon2 hasher = argon2.PasswordHasher(
          salt len=16 # defaults to 16
      hashed password = argon2 hasher.hash(password.decode('utf-8'))
      print("Argon2 hash", hashed password)
      password is right = argon2 hasher.verify(hashed password, "password")
      print("Password is right ", password is right)
          argon2 hasher.verify(hashed password, "wrong password")
      except argon2.exceptions.VerifyMismatchError:
          print("Password is wrong")
```

#### Checksum hashing algorithms ADLER-32 (MARK ADLER'S 32-BIT)

```
Run Cell | Run Above | Debug Cell

# %%

import zlib

file_checksum = 0

with open("../files/test_file.txt",'rb') as f:

while True:

chunk = f.read(4096) # Read 4KB at a time

if not chunk:

break

file_checksum = zlib.adler32(chunk)

print(f"hash of 'test_file.txt': {file_checksum}")

Run Cell | Run Above | Debug Cell

# %%
```

#### Password hashing algorithms TRADEOFFS

| CRITERIA              | CRC32                      | Adler-32                        |
|-----------------------|----------------------------|---------------------------------|
| Speed                 | Slower                     | Faster                          |
| Checksum Size         | 32 bits                    | 32 bits                         |
| Collision Resistance  | Better                     | Worse                           |
| Data Size Sensitivity | Performs well on all sizes | Performs poorly on small inputs |

#### thanks, best regards, Anderson Babinski





github.com/andeerlb



linkedin.com/andersonbabinski

This is a repo linked this slide github.com/andeerlb/hashing-algorithms