Project Documentation: Hash Cracker

Project Title:

Hash Cracker using Brute Force and Wordlist

Objective:

To crack a given hash using either a predefined wordlist or brute-force generated passwords. The tool supports multiple hash algorithms and uses multithreading for faster computation.

Technologies Used:

• Python 3
• hashlib – for hashing algorithms
• itertools, string – for password generation
• concurrent.futures.ThreadPoolExecutor – for multithreading
• tqdm – for progress visualization
• argparse – for command-line interface

Supported Hash Types:

md5, sha1, sha224, sha256, sha384, sha512, sha3_224, sha3_256, sha3_384, sha3_512

How It Works:

1. Wordlist Mode:
   - Takes a path to a wordlist file.
   - Compares each word (after hashing) against the target hash.
   - Stops and returns if a match is found.

2. Brute-force Mode:
   - Generates passwords with combinations of user-specified characters between min_length and max_length.
   - Hashes each generated password and checks for a match.
   - Multithreaded execution for speed.

Code Structure:

• generate_passwords(min_length, max_length, characters): Generator for passwords.
• check_hash(hash_fn, password, target_hash): Verifies if password hash matches the target.
• crack_hash(...): Core function that either uses a wordlist or brute-forces the hash.
• argparse section: Handles command-line input and initiates the crack.

Usage:

python hash_cracker.py <HASH> [options]

Examples:

• Using a wordlist:
python hash_cracker.py e99a18c428cb38d5f260853678922e03 -w rockyou.txt --hash_type md5

• Using brute-force:
python hash_cracker.py e99a18c428cb38d5f260853678922e03 --hash_type md5 --min_length 1 --max_length 4 -c abc123

Project Features:

• Multithreading for performance boost.
• Real-time progress tracking using tqdm.
• Flexible command-line interface.
• Supports both offline cracking modes (wordlist and brute-force).
• Easy to extend for future hash types.

Future Improvements:

• Add GPU support for faster cracking (e.g., via hashcat).
• Add dictionary merging and mutation strategies.
• Include rainbow table or salting options.
• Web-based GUI for usability.

Overview of the Hash Cracker Process

This Python script is a hash cracking tool that supports two modes:

1. **Dictionary Attack** (using a wordlist)
2. **Brute-Force Attack** (generating combinations)

It uses **multithreading** for faster execution and supports hash algorithms like **MD5, SHA1, SHA256, SHA3**, etc.

1. Library Imports

```
import hashlib
import itertools
import string
from concurrent.futures import ThreadPoolExecutor
from tqdm import tqdm
import argparse
```

- `hashlib`: For hashing algorithms
- `itertools`: For generating combinations
- `ThreadPoolExecutor`: For multithreading
- `tqdm`: For progress bar
- `argparse`: For CLI support

2. Supported Hash Types

```
hash_name = [
    'md5',
    'sha1',
    'sha224',
    'sha256',
    'sha384',
    'sha3_224',
    'sha3_256',
    'sha3_384',
    'sha3_512',
    'sha512'
]
```

Used to validate user inputs and avoid unsupported algorithms.

3. Password Generator

```
def generate_passwords(min_length, max_length, charcaters):  1 usage
    for length in range(min_length, max_length + 1):
        for pwd in itertools.product(charcaters, repeat=length):
            yield ''.join(pwd)
```

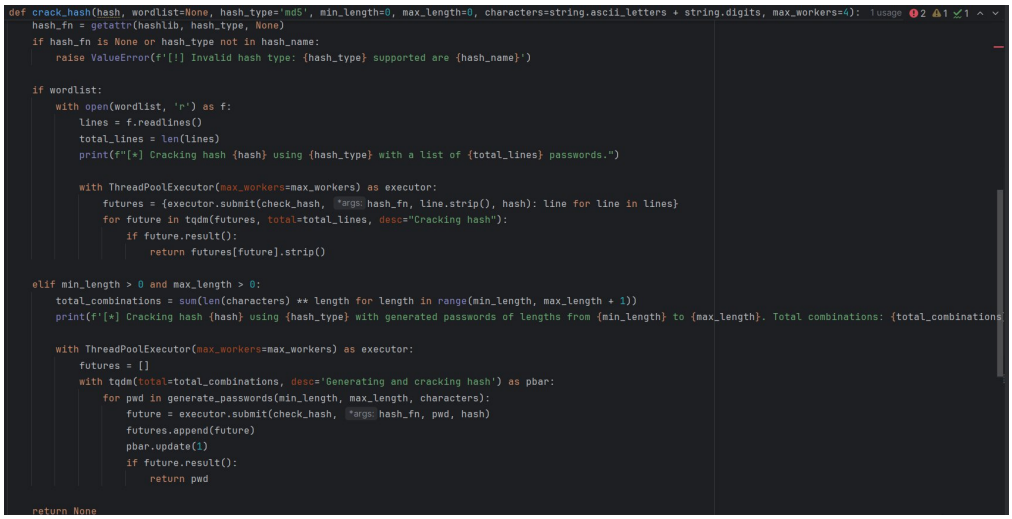
Creates all combinations (e.g. `abc` → `a`, `b`, `c`, `aa`, `ab`, ...)

4. Hash Matching

```
def check_hash(hash_fn, password, target_hash):  2 usages
    return hash_fn(password.encode()).hexdigest() == target_hash
```

Compares hashed password with target hash.

5. Main Cracking Logic



```
def crack_hash(hash, wordlist=None, hash_type='md5', min_length=0, max_length=0, characters=string.ascii_letters + string.digits, max_workers=4):  1 usage
    hash_fn = getattr(hashlib, hash_type, None)
    if hash_fn is None or hash_type not in hash_name:
        raise ValueError(f'[!] Invalid hash type: {hash_type} supported are {hash_name}')

    if wordlist:
        with open(wordlist, 'r') as f:
            lines = f.readlines()
            total_lines = len(lines)
            print(f"[*] Cracking hash {hash} using {hash_type} with a list of {total_lines} passwords.")

            with ThreadPoolExecutor(max_workers=max_workers) as executor:
                futures = {executor.submit(check_hash, *args: hash_fn, line.strip(), hash): line for line in lines}
                for future in tqdm(futures, total=total_lines, desc="Cracking hash"):
                    if future.result():
                        return futures[future].strip()

    elif min_length > 0 and max_length > 0:
        total_combinations = sum(len(characters) ** length for length in range(min_length, max_length + 1))
        print(f'[*] Cracking hash {hash} using {hash_type} with generated passwords of lengths from {min_length} to {max_length}. Total combinations: {total_combinations}

        with ThreadPoolExecutor(max_workers=max_workers) as executor:
            futures = []
            with tqdm(total=total_combinations, desc='Generating and cracking hash') as pbar:
                for pwd in generate_passwords(min_length, max_length, characters):
                    future = executor.submit(check_hash, *args: hash_fn, pwd, hash)
                    futures.append(future)
                    pbar.update(1)
                    if future.result():
                        return pwd

    return None
```

a) Dictionary Attack:

- Reads wordlist
- Threads check each password
- Stops on match

b) Brute-force Attack:

- Dynamically generates combinations
- Threads evaluate them in parallel

6. Command Line Interface (CLI)

Takes user input via:

- `--hash_type`
- `--wordlist`
- `--min_length`, `--max_length`
- `--characters`, `--max_workers`

7. Sample Execution

Wordlist mode:

```
python hash_cracker.py <hash> -w rockyou.txt --hash_type md5
```

Brute-force mode:

```
python hash_cracker.py <hash> --hash_type md5 --min_length 1 --
max_length 4 -c abc123
```

Output :-