Practical 1 COS330

Hayley Dodkins

July 2025

System Specifications

```
Hardware Overview:

  Model Name: MacBook Pro
  Model Identifier: MacBookPro18,3
  Model Number: MKGP3ZE/A
  Chip: Apple M1 Pro
  Total Number of Cores: 8 (6 performance and 2 efficiency)
  Memory: 16 GB
  System Firmware Version: 11881.121.1
  OS Loader Version: 11881.121.1
  Serial Number (system): V402XP9JKN
  Hardware UUID: 7C8389A4-C9F9-504E-A337-D0896E26FE6B
  Provisioning UDID: 00006000-000409D814FA401E
  Activation Lock Status: Enabled
```

1 Task 1

1.2 Hashing Algorithm Choices

I used the following three hashing algorithms:

- **MD5:** The MD5 hashing algorithm was declared cryptographically broken in 2008. One of the largest problems with the MD5 algorithms is that there is a likely chance of collisions, which is when distinct inputs produce the same output Okta (2024).
- **SHA-256:** A hashing algorithm that produces a value that is 256 bits long. SHA-256 is one of the more secure hashing algorithms, notably it is used in SSL, TLS and PGP. Unix and Linux systems use SHA-256 as the password hashing algorithm N-able (2019). The algorithm is secure due to the fact that a brute force attack would need 2^{256} attempts to get the input, collisions are very unlikely and a minor change the the input will lead to a drastically different output, this is known as the avalanche effect and it is very desirable for cryptographic algorithms Law (N/A).

- **bcrypt:** This hashing algorithm was design to be strong against brute force attacks making it a great choice as a password hashing algorithm. Unlike SHA-256 and MD5, bcrypt includes a salt to provide more security. Bcrypt is iterative meaning it takes longer to verify and generate a hash but this also makes brute force attacks harder since each guess takes a long time Bhupendra (2024).

1.3 Database Schema

The table `users` was extended to include three new columns: `md5_hash`, `sha256_hash`, and `bcrypt_hash`, in addition to the original `id`, `username`, and `password` fields. Figure 1 shows the schema of the users table, and Figure 2 shows the first five rows after hashing.

```
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.
sqlite> .schema users
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL
, md5_hash TEXT, sha256_hash TEXT, bcrypt_hash TEXT);
```

Figure 1: Database schema of the `users` table

```
sqlite> SELECT * FROM users LIMIT 5;
1|david.davis50|gIMrC|6ea8939892ec9a10662758 4ee83a6ef3|9498a3566479 5512ced8cd78efc5dd3a2a55a69d2bac3fd498c325d2db0f26f6|$2a$12$S.ApfNzcXNRevQHOG1INxeV0qm6VWKxJz97V/EUibgWM03nv7cyLi
2|sara.smith47|passwor|f88d9aaa52161c2e4f8c42c0d86aa05d|2a2f822ad5319117db5816955c0d89d98a59e5886789cf8883fc6abdd2a1e17|$2a$12$dktXgvCcKseOHf4FuJU68u/EUuj59j8Lp8XSDWx68AJr8i/QA1yXO
3|john.davis94|srdu|ea426ec1bf87a552a7e1237b3cc2cb62|315a83e8b2de01e04a36a3aebd3482d773b980344fa536262b4d185fa63c5ef|$2a$12$QjepBQeVqLTM6lyLEuXveurrs3e3dnZ6hkSHKbcF7v6L86fnvvU2
4|alex.davis54|enjI1O|001b5875b4a324cd81b0007f3fa45f95|57beed2eff7d83d58b24049422ce8044859ee662b4e7f383a0fffbcfd3cf89b4|$2a$12$o.TQcN0ULThw0a/01Hn4uOhYLJ3prs8VSz1CeUaxE14GU4kkO5ZEK
5|john.davis6|Ii1G|d715bedfb0c7c21e8cfeade4eca5e0a7|c411d879ff0fe4ef2322c588a71c44bf3dbfc753fa45dda85915782cce4ab824|$2a$12$/KZbnzXTbwEoeW36A56ca08VSnfmeYBehE0pyHMOAbnFXZLayJfDi
```

Figure 2: First 5 rows of the `users` table

2 Task 2

2.1 Methodology

I performed dictionary attacks on the hashed passwords in the SQLite database using two approaches:

- **John the Ripper:** I used John the Ripper, a popular password cracking tool, to perform dictionary attacks using multiple word lists.
- **Manual Ruby implementation:** I designed and implemented a custom SHA-256 password cracker written in Ruby to iterate through word lists and compare digests manually, stopping after 3 minutes if no match was

found. I chose to stop the run after 3 minuets because it would take a long time to iterate through all the passwords and all the words in the word list.

I tested all three hashing algorithms (MD5, SHA256, bcrypt). However, bcrypt was excluded from cracking due to its resilience to brute force attacks and salting, which renders dictionary attacks impractical without prior knowledge of the salt.

2.2 Commands Used

The following commands were executed using John the Ripper to perform dictionary attacks against the MD5 and SHA-256 hashes extracted from the database. Each set of commands corresponds to a specific wordlist.

```
john --format=raw-md5 hashes/md5_hashes.txt --wordlist=utils/rockyou.txt
john --format=raw-SHA256 hashes/sha256_hashes.txt --wordlist=utils/
    rockyou.txt
```

```
john --format=raw-md5 hashes/md5_hashes.txt --wordlist=utils/
    darkweb2017_top-10000.txt
john --format=raw-SHA256 hashes/sha256_hashes.txt --wordlist=utils/
    darkweb2017_top-10000.txt
```

```
john --format=raw-md5 hashes/md5_hashes.txt --wordlist=utils/scraped-JWT-
    secrets.txt
john --format=raw-SHA256 hashes/sha256_hashes.txt --wordlist=utils/
    scraped-JWT-secrets.txt
```

Displaying Cracked Hashes

After running the cracking jobs, the following commands were used to display all cracked hashes. The `--show` option outputs the associated user ID and recovered plaintext password without re-running the attack.

```
john --show --format=raw-md5 hashes/md5_hashes.txt
john --show --format=raw-SHA256 hashes/sha256_hashes.txt
```

2.3 Results Summary

View ruby code

To automate the cracking process and compare results across hashing algorithms and wordlists, I implemented a custom Ruby script using `'sqlite3'`, `'open3'`, and process forking. First, I extracted all non-null `md5_hash` and `sha256_hash` values from the `users_realistic.db` SQLite database into format-specific hash files. For MD5, hashes were prefixed with a dynamic format string required by John the Ripper. Then, I used Ruby's 'Open3.capture3' to execute cracking jobs with John the Ripper for each hash type using three different wordlists: `rockyou.txt`, `darkweb2017_top-10000.txt`, and `scraped-JWT-secrets.txt`.

Each pair of MD5 and SHA-256 cracking tasks were executed in parallel using 'Process.fork', significantly speeding up overall runtime by utilizing multiple CPU cores. Once completed, the script collected cracked results using `'john --show'`, stored them in JSON files for later use, and computed the total runtime and average time per cracked hash. Final results were printed to the console with colored formatting for better readability.

This pipeline allowed for an efficient, repeatable process to benchmark cracking effectiveness across different wordlists and hash types, while also demonstrating scripting and automation proficiency in Ruby.

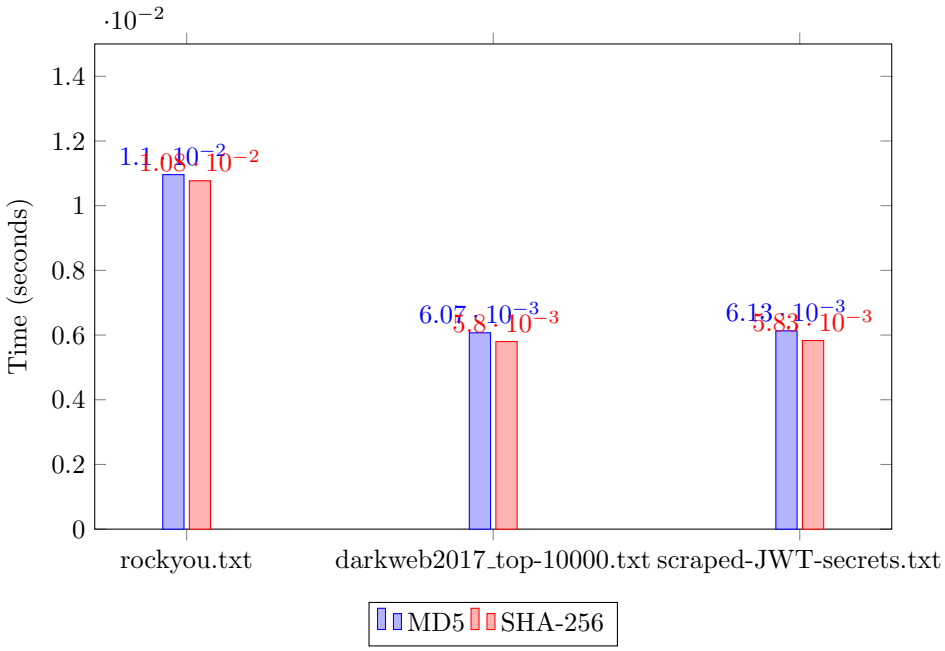
Figure 3: John the Ripper cracking output for MD5 and SHA256

2.4 Performance Comparison

Table 1: Average time per cracked hash (in seconds) across different iterations and wordlists

Iteration	**rockyou.txt**		**darkweb2017_top-10000.txt**		**scraped-JWT-secrets.txt**	
	MD5	**SHA-256**	**MD5**	**SHA-256**	**MD5**	**SHA-256**
1	0.0122	0.0121	0.0060	0.0058	0.0060	0.0058
2	0.0095	0.0092	0.0062	0.0058	0.0065	0.0062
3	0.0112	0.0110	0.0060	0.0058	0.0059	0.0055

Figure 4: Average Cracking Time per Hash (in seconds)



Discussion: The table above presents the average time taken to crack individual hashes using three different wordlists across multiple iterations. The time taken to crack each password hash varied across wordlists and hash algorithms. These variations are influenced by multiple factors:

- **Wordlist Size and Order:** Larger wordlists like `rockyou.txt` contain millions of entries and take longer to iterate through. Smaller wordlists like `darkweb2017_top-10000.txt` offer faster lookups due to fewer entries.
- **Hash Complexity:** SHA-256 involves more complex computations compared to MD5, which can slightly increase per-hash processing time.
- **System Load and Parallel Jobs:** Since hash cracking was run in parallel, system resources like CPU cores were shared, occasionally causing performance fluctuations between runs.
- **Early Matches:** In cases where a matching password was found early in the wordlist, the cracking process terminated sooner, resulting in a faster observed average.
- **Wordlist Size and Order:** The size and ordering of the wordlist impacts performance. Larger lists like `rockyou.txt`, which contains over 14 million entries, take longer to process due to the sheer volume of words.

In contrast, smaller lists such as `darkweb2017_top-10000.txt` are optimized for likelihood and speed, containing fewer but more commonly used passwords, which leads to faster results in many cases.

- **Hash Complexity:** Different hashing algorithms require varying amounts of computation. SHA-256, being a more modern and secure algorithm than MD5, performs more rounds of internal operations, which make it slightly slower per hash comparison.
- **System Load and Parallel Jobs:** To speed up the cracking process, multiple cracking jobs were forked and run in parallel. While this takes advantage of multi-core processors, it also leads to increased contention for shared resources such as CPU time and memory bandwidth. Depending on what other background processes were running, this may have caused inconsistent performance between iterations.
- **Early Matches:** John the Ripper stops iterating through a wordlist once a match is found for a given hash. If the matching password appears early in the list, the average time to crack is significantly reduced. This variability means even across multiple runs with the same setup, some iterations can appear faster due to the position of the correct password in the list.
- **Caching and Disk I/O Effects:** Repeated runs using the same wordlists may benefit from operating system level disk caching. Once a file is loaded into memory, subsequent accesses are faster, which can improve performance on later iterations. Conversely, high I/O demand can slow down runs if the wordlist has to be reloaded from disk.

These timing differences highlight how both the structure of the wordlist and the efficiency of the hash algorithm affect brute-force performance.

2.5 Optimization and Estimation

To optimize cracking I did the following:

- Hashes were separated into format-specific files
- Cracking tasks for different algorithms were run in parallel using separate processes to reduce total runtime.
- Multiple wordlists were tested, including `rockyou.txt`, `darkweb2017_top-10000.txt`, and `scraped-JWT-secrets.txt`, to increase the likelihood of successful matches.
- For the custom implemented SHA-256 password cracker, a 3 minute execution limit was introduced to avoid long execution times.

2.6 Custom Cracking Implementation

View custom ruby cracker implementation

```
hayleydodkins@MacBook-Pro-von-Hayley practical1 % ruby manual_hash_cracker.rb
Found
Found
Found
Found
Spent over 3 minutes looking
```

Figure 5: Custom SHA256 cracking output in Ruby

While the custom implementation successfully demonstrates a brute-force dictionary attack using SHA256 in Ruby, it performs significantly slower than tools like John the Ripper. This is expected since John is written in highly optimized C, which supports multi-threading and SIMD instructions. In contrast, the Ruby script processes each user and each password sequentially, using interpreted code with no hardware acceleration or parallelization. Additionally, Ruby's Digest::SHA256 and file I/O operations are relatively slow, especially when processing large dictionaries. The goal of this implementation was not to show speed, but rather to demonstrate an understanding of the underlying brute-force process and to show how easily hashes can be cracked when weak passwords are used.

3 Task 3

Login Functionality Code

The following Ruby function demonstrates how a user's login would be verified. View Login Code File

```ruby
require 'rubygems'
require 'bcrypt'
require 'openssl'
require 'base64'
require 'securerandom'

PEPPER = ENV['PEPPER'] || "supersecretpepper"

def encrypt_salt(salt, key)
  cipher = OpenSSL::Cipher.new('AES-256-CBC')
  cipher.encrypt
  cipher.key = key
  iv = cipher.random_iv
  cipher.iv = iv
  encrypted = cipher.update(salt) + cipher.final
  Base64.encode64(iv + encrypted)
end

def decrypt_salt(encrypted_salt_b64, key)
  encrypted_salt = Base64.decode64(encrypted_salt_b64)
  cipher = OpenSSL::Cipher.new('AES-256-CBC')
  cipher.decrypt
  cipher.key = key
  cipher.iv = encrypted_salt[0...16]
  cipher.update(encrypted_salt[16..]) + cipher.final
end

def register_user(plain_password, aes_key)
  salt = SecureRandom.hex(8)
  encrypted_salt = encrypt_salt(salt, aes_key)
  full_password = plain_password + salt + PEPPER
  hashed_password = BCrypt::Password.create(full_password)

  {
    encrypted_salt: encrypted_salt,
    hashed_password: hashed_password.to_s
  }
end

def verify_login(input_password, user_record, aes_key)
  salt = decrypt_salt(user_record[:encrypted_salt], aes_key)
  full_password = input_password + salt + PEPPER
  hashed = BCrypt::Password.new(user_record[:hashed_password])
  hashed == full_password
end


aes_key = Digest::SHA256.digest("secret_aes_password")
user_record = register_user("mySecurePass123", aes_key)
```

```
puts "Login attempt with correct password:"
puts verify_login("mySecurePass123", user_record, aes_key) ? " Success" :
     "Failure"

puts "\nLogin attempt with wrong password:"
puts verify_login("wrongPassword", user_record, aes_key) ? "Success" : "
    Failure"
```

How It Works

- **Step 1:** When a user registers, a random salt is generated and encrypted using AES with a secret key.
- **Step 2:** The plaintext password is combined with the salt and pepper, then hashed using bcrypt.
- **Step 3:** During login, the stored encrypted salt is decrypted, and the input password is rehashed the same way. The hashes are compared for equality.

Justification of Method

Salt Generation: A random salt is generated for each user using SecureRandom.hex(8). This ensures that even if two users have the same password, the input to the hash function will differ, making rainbow table attacks ineffective Lin (2016).

Salt Encryption: To encrypt the salt in Ruby, I used the OpenSSL gem. I begin by creating a new cipher instance using `AES-256-CBC`. The Advanced Encryption Standard (AES) is a symmetric block cipher. AES-256 uses 14 rounds of transformation and a 256-bit key, making it extremely secure. The longer the key, the more combinations an attacker must try in a brute-force attack, with AES-256, there are 2^{256} possibilities, making such attacks infeasible Kiteworks (N/A). From the available AES modes, I chose **Cipher Block Chaining (CBC)**. In CBC mode, each plaintext block is XORed with the previous ciphertext block before encryption. For the first block, a random **Initialization Vector (IV)** is used instead. This chaining mechanism ensures that identical plaintext inputs result in different ciphertexts, given that the IVs are different. While CBC requires managing IVs correctly and applying padding, these are reasonable trade-offs for the added security. CBC ensures that if two users have the same password and salt, the resulting encrypted salt will differ due to the random IV Kiteworks (N/A). When encrypting the salt, I generate a random IV and prepend it to the ciphertext. This IV is then used during decryption to correctly reverse the process. The salt is encrypted using the `encrypt_salt` method. Encrypting the salt adds an additional layer of security. Even if an

attacker gains access to the database, they would still need the AES key to retrieve the salts, making it more difficult to perform brute-force or offline attacks.

Peppering: A static pepper is stored securely in an environment variable and appended to the password and salt combination. Unlike salts, the pepper is not stored in the database. This provides added security against dictionary attacks. This means even if the database is compromised, the attacker cannot compute the full input to the hash function without also knowing the pepper OWASP (N/A) Lin (2016).

Password Hashing: The combined string of password + salt + pepper is hashed using BCrypt::Password.create. Bcrypt is a computationally expensive, adaptive hashing algorithm designed specifically for password storage. Its slowness works as a defense mechanism, making brute-force attacks more costly.

Login Verification: During login, the encrypted salt is first decrypted using the AES key. The input password is concatenated with the salt and pepper, and the resulting string is verified using bcrypt's == comparison operator. This ensures that only a correct password will match the stored bcrypt hash.

References

Bhupendra (2024), *Password Hashing using bcrypt*, `https://medium.com/@bhupendra_Maurya/password-hashing-using-bcrypt-e36f5c655e09`. Accessed: 28 July 2025.

Kiteworks (N/A), *Everything You Need to Know About AES-256 Encryption*, https://www.kiteworks.com/risk-compliance-glossary/aes-256-encryption/. Accessed: 28 July 2025.

Law, F. (N/A), *What You Need To Know About the "Avalanche" Effect*, `https://freemanlaw.com/what-you-need-to-know-about-the-avalanche-effect/`. Accessed: 28 July 2025.

Lin, R. (2016), *Salt Pepper: Spice up your hash!*, `https://medium.com/@berto168/salt-pepper-spice-up-your-hash-b48328caa2af`. Accessed: 28 July 2025.

N-able (2019), *SHA-256 Algorithm Overview*, `https://www.n-able.com/it/blog/sha-256-encryption`. Accessed: 28 July 2025.

Okta (2024), *What is MD5? Understanding Message-Digest Algorithms*, `https://www.okta.com/identity-101/md5/`. Accessed: 28 July 2025.

OWASP (N/A), *Password Storage Cheat Sheet*, https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. Accessed: 28 July 2025.