Karima Newman

Computer Security – 3721

Nov 11, 2025

My project simulates offline dictionary attacks to evaluate how different password-hashing algorithms perform under a range of configurations. The simulation measures the probability of successful password cracking along with hash computation time to compare weak legacy hashing methods against modern, parameter-tunable key-derivation functions. Specifically using MD5, SHA-1, SHA-256, PBKDF2, bcrypt, and Argon2 while varying salting, dictionary size, and algorithm parameters including iteration count, memory, and cost factors. By comparing both cracking effectiveness and performance impact, this project identifies best practices for implementing strong password storage schemes.

The overall goal of the project is to gain insight into how password hashing algorithms differ in real defense value. More specifically, the project aims to:

1. Evaluate the safety of multiple hashing algorithms under offline cracking.
2. Measure the impact of salting on preventing successful reverse-engineering.
3. Understand how algorithm parameters affect cracking success.
4. Compare outcomes across dictionary sizes to approximate attacker capabilities.
5. Provide practical recommendations for secure password storage design.

Attackers often obtain password databases through system compromise. When these databases contain hashed not plaintext passwords, attackers attempt to reverse them using dictionary attacks or brute force. Weak or fast hashing algorithms are extremely vulnerable

because attackers can compute billions of guesses per second. Modern, memory-hard, or cost-parameterized algorithms deliberately make hashing computationally expensive, greatly slowing attackers while remaining manageable for authentication systems. Understanding the differences in resistance helps guide appropriate choice and configuration of password hashing systems in real deployments.

The system is composed of six major components working together to simulate password storage and cracking:

1. Password Generator

   This module produces a corpus of approximately 2,000 randomly generated passwords of varying strength, including weak, medium, and strong samples.

2. Dictionary Source

   Two dictionary wordlists are used to simulate different adversary capabilities. The first is a small list of common passwords. The second is a large list representing more sophisticated attacker guess sets.

3. Hashing Engine

   The hashing engine takes each password and applies a hashing algorithm with optional salt and algorithm-specific parameters. Tested algorithms include MD5, SHA-1, SHA-256, PBKDF2, bcrypt, and Argon2.

4. Cracking Simulator

   During simulation, each dictionary word is hashed and compared to the stored hash. If a match occurs, the password is considered cracked.

5. Results

   The simulator records cracked status, password strength, hash algorithm, parameter choices, dictionary used, and measured hash time. Results are saved into CSV files for analysis.

6. Visualization

   Using matplotlib, the system generates graphs illustrating cracking rate vs. algorithm, hashing time vs. algorithm, and heatmaps showing parameter influence.

Passwords are generated and passed into the hashing engine with salting and algorithm parameters. Hashes are stored. Next, dictionary words are passed to the cracking simulator, which attempts to match hashes. Results are collected into CSVs and graphs.

Passwords → Hash Engine → Stored Hashes

Dictionary → Cracking Simulator → Match? → Results → CSV + Plots

The simulation is implemented entirely in Python 3. The following major libraries are used:

- hashlib, bcrypt, and argon2-cffi for hashing
- pandas and numpy for data storage and analysis
- matplotlib for visualization
- time for timing hash computation

No external APIs are required, the system runs right on any machine with Python installed.

The simulation varies parameters for relevant algorithms:

- PBKDF2: 1,000; 10,000; and 50,000 iterations
- bcrypt: cost 8, 10, and 12
- Argon2: 32 KB, 256 KB, and 1,024 KB memory

All algorithms are evaluated with and without salting. Each configuration is run across both dictionary sizes.

To run the system, install Python 3 and the libraries.

Install dependencies:

pip install pandas numpy matplotlib bcrypt argon2-cffi

To run the main simulation:

python simulate_hashing_project.py

This produces a directory called results/ containing:

- results.csv — detailed per-password outcomes
- summary.csv — aggregated statistics
- hash_time_by_algo.png — hashing performance graph
- cracked_rate_by_algo.png — cracking success graph

Faster cryptographic hashes yield the highest cracking success rates. Slower KDFs perform significantly better. Approximate cracking rates observed:

Algorithm	Cracked Rate
MD5	~66–92%
SHA-1	~64–89%
SHA-256	~60–80%
PBKDF2	17–63% depending on iteration count
bcrypt	9–39% depending on cost
Argon2	4–20% depending on memory

The results confirm that traditional hashes (MD5, SHA-1, SHA-256) provide inadequate security against offline attacks, even with salting.

Salting consistently reduced cracked rates across all algorithms by approximately 20–40%. Salts are essential because they render precomputed attacks ineffective

Increasing cost parameters lowered the cracked rate dramatically. Examples:

PBKDF2

Iterations	Crack Rate
1,000	~51–64%

Iterations	Crack Rate
10,000	~27–52%
50,000	~17–32%

bcrypt

Cost	Crack Rate
8	~32–39%
10	~15–30%
12	~9–19%

Argon2

Memory	Crack Rate
32 KB	~9–20%
256 KB	~6–15%
1,024 KB	~4–10%

This demonstrates the strong security benefit of increasing the hashing workload.

The simulation data strongly supports the conclusion that MD5, SHA-1, and SHA-256 are not sufficiently resistance to offline cracking. PBKDF2 provides reasonable resistance when configured with high iteration counts. bcrypt provides strong adaptive protection tied to its cost

parameter. Argon2 consistently performs best due to its memory hardness, further hindering parallelized cracking.

Salting universally improves outcomes. Parameter tuning is essential and must be revisited regularly to keep pace with hardware advancements.

In conclusion, this simulation demonstrates how choice of password-hashing algorithm, salting, and parameter tuning drastically impact recoverability under offline attack. Traditional cryptographic hashes fail to provide adequate protection, whereas PBKDF2, bcrypt, and especially Argon2 significantly reduce cracking success. Password security must combine strong algorithms, salts, and cost parameters to maintain robust defense.