PASSWORD HASHING BENCHMARKING

Evaluation of ARGON2, BCRYPT, and PBKDF2

Robert Gravelle
July 2025

Abstract:
This report benchmarks and analyzes the performance, resource usage, and cryptographic properties of three widely-used password hashing algorithms—Argon2id, bcrypt, and PBKDF2—to guide secure password storage decisions in modern systems.

Executive Summary + Introduction

1. Overview

This report presents a comparative benchmark of three widely-used password hashing algorithms: **Argon2id**, **bcrypt**, and **PBKDF2-HMAC-SHA256**. These functions are foundational to password-based authentication systems and determine how securely user credentials are stored.

2. Purpose

The goal of this benchmark is to measure:

- Average time required to hash passwords at various cost parameters
- Memory usage during hashing
- Output hash lengths and format
- Trade-offs between performance and security

3. Why This Matters

Poorly chosen hashing strategies make systems vulnerable to brute-force and GPU-based dictionary attacks. Evaluating algorithmic cost, speed, and memory requirements helps inform best practices for securely storing passwords.

Technical Background

1. Hashing Algorithm Overview

Algorithm	Description
Argon2id	Memory-hard password hashing algorithm designed to resist GPU attacks. Winner of the Password Hashing Competition (PHC).
bcrypt	Key derivation function based on the Blowfish cipher. Designed for incremental cost adjustment through "rounds."
PBKDF2	Password-Based Key Derivation Function 2, standardized in RFC 2898. Uses repeated application of HMAC-SHA256.

2. Security Features

Feature	Argon2id	bcrypt	PBKDF2
Adjustable Time Cost	✅	✅	✅
Adjustable Memory Cost	✅	❌	❌
Parallelism Control	✅	❌	❌
Built-in Salt Handling	✅	✅	❌ (manual)
Resistance to GPU/ASIC	✅ Strong	Moderate	Weak

Benchmarking Methodology

1. System Setup

- **Platform**: Windows 11
- **CPU**: Intel Core i5-12400 (Intel64 Family 6 Model 154)
- **RAM**: 16 GB DDR4
- **Python Version**: 3.x (64-bit)
- **Hashing Libraries**: argon2-cffi, bcrypt, psutil, hashlib
- **Repetition:** Each benchmark was run 10 times and averaged for accuracy
- Password: "correcthorsebatterystaple"

2. Metrics Collected

- **Average time (seconds)** to hash a single password
- **Memory used (MB)** during hashing (measured via psutil)
- **Hash output length**, in characters

  Note: Memory usage was measured using psutil, which tracks *resident set size (RSS)*. This may not reflect peak memory used internally by native libraries (e.g., Argon2id's C-level allocations). Despite increasing memory_cost to 256MB, psutil reported minimal change — expected behavior due to Python's memory tracking limitations.

3. Test Ranges

- Argon2id: time_cost = [2, 3, 4]
- bcrypt: rounds = [12, 13, 14]
- PBKDF2: iterations = [600k, 800k, 1M]

Results & Interpretation

Benchmark Summary (10-run average)

Memory Cost (KB)	Avg Time (s)	Output Length	RSS Delta (MB)
65536 (64 MB)	0.0610	97	0.01
131072 (128 MB)	0.1155	98	0.00
262144 (256 MB)	0.2363	98	0.00

Argon2id — Varying Parallelism

Threads	Avg Time (s)	Output Length	RSS Delta (MB)
1	0.1615	97	0.00
2	0.0992	97	0.00
4	0.0607	97	0.00
8	0.0503	97	0.00

bcrypt & PBKDF2 Summary

Algorithm	Cost Param	Avg Time (s)	Output Length	RSS Delta (MB)
bcrypt	rounds=12	0.3139	60	0.00
bcrypt	rounds=13	0.6255	60	0.00
bcrypt	rounds=14	1.2555	60	0.00
PBKDF2	600,000 iter	0.3189	97	0.01
PBKDF2	800,000 iter	0.5577	97	0.00
PBKDF2	1,000,000 iter	0.9729	97	0.00

Observations:

- **Argon2id** shows excellent scalability, improving performance with both increased threads and tunable memory cost.
- **bcrypt** performs predictably but scales exponentially with rounds, making high-cost settings impractical on many systems.
- **PBKDF2** demonstrates linear time scaling with iterations but lacks memory hardness and must be paired with secure, unique salts for safety.

Recommendations & Conclusions

1. Recommended Use

Scenario	Best Choice
New system with modern hardware	✅ **Argon2id** (default for new apps)
Compatibility with older systems	✅ **bcrypt**
Compliance with NIST standards	✅ **PBKDF2**

2. Key Takeaways

- Always adjust cost parameters to ensure **at least 100ms** hashing time on target hardware.
- Use **memory-hard algorithms** like Argon2id when GPU/ASIC attacks are a concern.
- Avoid fixed salts or low iteration counts in production.
- Benchmark regularly as hardware evolves.
- When using PBKDF2, developers must manually supply a unique random salt for each password. Reusing salts makes systems vulnerable to rainbow table and pre-image attacks.

3. Final Thoughts

Password hashing is a critical defense against credential theft. This benchmark demonstrates the trade-offs and performance of three leading algorithms and can serve as a foundation for securing authentication systems with informed, data-driven decisions.