ISSN 2320 - 2602
Volume 14 No. 7, July 2025
International Journal of Advances in Computer Science and Technology
Available Online at http://www.warse.org/IJACST/static/pdf/file/ijacst011472025.pdf
https://doi.org/10.30534/ijacst/2025/011472025

The Role of Hashing Libraries in Flask Application Security: A Focus on Password Protection

Lord Ian M. Paquiao[1], Zoren E. Bajao[2]
[1]ACLC College of Butuan, Philippines, lordianpaquiao@gmail.com
[2]ACLC College of Butuan, Philippines, 02zorenbajao@gmail.com

Received Date : May 21, 2025 Accepted Date : June 28, 2025 Published Date : July 07, 2025

ABSTRACT

Flask-based web apps still require password protection, particularly as threat landscapes change. Werkzeug.security, Flask-Bcrypt, Flask-Argon2, and Passlib are popular Flask-compatible password hashing libraries. They were compared in terms of cryptographic strength, default security settings, hashing latency, system resource consumption, and ease of integration using experimental benchmarking. Tests were conducted in a controlled Flask environment using standardized profiling tools and simulated user interactions. Flask-Bcrypt outperformed the other libraries by balancing developer-friendly integration, reasonable latency, and robust security defaults. Although Flask-Argon2 had strong cryptographic protection, it used a lot of CPU and memory. Although Werkzeug.security was effortless and performed well, it must be manually configured to satisfy security requirements. Although Passlib had the most integration complexity, it was notable for its configurability. The results emphasize the significance of selecting hashing tools based on implementation feasibility and algorithm strength. When choosing password hashing options for Flask apps, developers are urged to consider usability, security, and performance.

Key words: Flask Framework, Password Hashing, Web Application Security, Cryptographic Security, Benchmarking

1. INTRODUCTION

Because of its adaptability, ease of use, and extensibility, Flask, a simple Python micro-framework, is frequently used for web development [1]. However, to achieve safe authentication, it mainly relies on third-party libraries. Strong password hashing techniques are essential for protecting user credentials during increasing data breaches. Werkzeug.security, Flask-Bcrypt, Flask-Argon2, and Passlib are popular Flask-compatible hashing libraries that are evaluated in this study based on their cryptographic strength, default security parameters, resource requirements, and ease of integration under practical deployment scenarios [2], [3].

According to earlier studies, strong password hashing methods are essential for modern web application security. Despite lacking memory-hardness, Bcrypt, which was first employed in the late 1990s, is still in use today because it can withstand brute-force attacks through modified cost factors [4], [5]. On the other hand, Argon2, the 2015 Password Hashing Competition winner, adds a memory-hard structure, which significantly increases resistance to parallelized attacks from ASICs and GPUs [6], [7]. According to one study, about 47% of open-source implementations employ configurations that are not OWASP-compliant, indicating that despite Argon2's advantages, its practical adoption is frequently incorrect [2]. To further improve password security, new technologies, such as multi-factor credential hashing, combine conventional hashing with extra entropy sources like hardware tokens or OTPs [4], [8].

Despite abundant research on cryptography, not much focus has been presented on the actual implementation of these libraries within particular frameworks, such as Flask. Developers often use defaults without fully comprehending their impacts, leading to vulnerabilities [3], [9]. It is common to ignore practical considerations such as documentation clarity, user load performance, and integration effort [6], [9].

To bridge this gap, this study benchmarks four widely adopted Flask-compatible hashing libraries—werkzeug.security, Flask-Bcrypt, Flask-Argon2, and Passlib—using criteria such as iteration cost, memory usage, response latency, and developer experience. The goal is to provide evidence-based recommendations for developers securing password authentication in Flask applications [2], [3], [10].

Although Flask is utilized in startup, enterprise, and academic settings, not much research has been done on Flask-specific password hashing techniques, which makes this study important. This research aims to enhance secure development methods and lower the risks of credential disclosure in Python-based web applications by providing a comparative assessment based on real-world situations [11], [12].

2. METHODOLOGY

This study employs a comparative experimental research design to evaluate the role of password hashing libraries in enhancing security within Flask-based web applications.

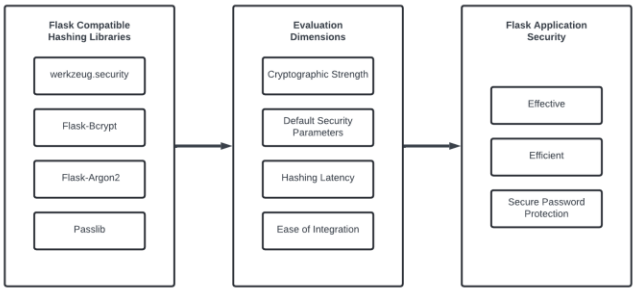

Figure 1: Conceptual Framework of the Study

Figure 1 presents the conceptual framework used in this study, illustrating the relationship between Flask-compatible hashing libraries and Flask application security. The four libraries evaluated—werkzeug.security, Flask-Bcrypt, Flask-Argon2, and Passlib—represent the independent variables. These libraries are assessed through four key evaluation dimensions: cryptographic strength, default security parameters, hashing latency, and ease of integration. These dimensions serve as intervening variables that influence the effectiveness and efficiency of password protection mechanisms within Flask applications. The dependent variable, Flask application security, is characterized by the system's ability to deliver secure, effective, and efficient password management. This framework guides the experimental benchmarking and comparative analysis conducted in the study.

2.1 Experimental Setup

Table 1: Test Environment

Test Environment	
Operating System	Windows 11 (64-bit)
Python Version	3.12.6
Flask Version	3.0.2
Hardware	Intel i5 10th Gen CPU, 16 GB RAM
Tools	Flask for web app simulation.
	timeit, psutil, and memory_profiler for benchmarking.
	Pandas/Matplotlib for analysis and visualization.

Table 1 shows the hardware and software setup used for the benchmarking tests. The experiments were conducted on a Windows 11 machine with Python 3.12.6, Flask 3.0.2, an Intel i5 processor, and 16 GB of RAM. Tools such as timeit, psutil, and memory_profiler were used to measure performance, while Pandas and Matplotlib were used for data analysis and visualization.

2.2 Evaluation Criteria

Table 2: Benchmarking Parameters

Criterion	Description
Cryptographic Strength	Resistance to brute-force and GPU/ASIC-based attacks.
Default Security Parameters	Cost factor, salt length, and alignment with OWASP recommendations.
Hashing Latency	Time taken to hash a password under various load conditions.
Ease of Integration	Developer experience in terms of setup, documentation, and code complexity.

Table 2 presents the key metrics used to benchmark the password-hashing libraries evaluated in this study. These include cryptographic strength, which refers to the library's ability to withstand brute-force and hardware-accelerated attacks; default security parameters, which examine settings such as cost factor and salt length about OWASP guidelines; hashing latency, which measures the time required to process password hashes under varying load conditions; and ease of integration, which reflects the developer experience in terms of implementation effort, documentation quality, and code complexity. These metrics provide a balanced framework for assessing each library's technical robustness and practical usability.

2.3 Benchmarking Procedures

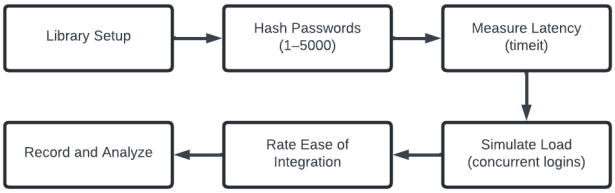

Figure 2: Step-by-Step Benchmarking Procedure

Figure 2 illustrates the step-by-step benchmarking workflow used in this study to evaluate the selected password hashing libraries. The process begins with integrating each library into a standard Flask application setup. Passwords are then hashed in increasing volumes to test performance under varying loads. Latency is measured using precise timing tools, while concurrent user requests are simulated to assess scalability. The ease of integration is also evaluated based on developer experience during setup and usage. Finally, all collected data—including performance metrics and usability notes—are recorded and analyzed to support the comparative evaluation. This workflow directly corresponds to the evaluation dimensions outlined in the conceptual framework, ensuring that each library's technical and practical aspects are systematically assessed.

2.4 Data Collection and Analysis

Table 3: Data Collection

Criterion	Description	Tool Used
Hashing Time (ms)	Measured duration per password hash.	time.perf_counter()
CPU Usage (%)	Monitored during hashing operations.	psutil.cpu_percent()
Memory Usage (MB)	Measured memory consumed by hashing processes.	memory_profiler
Load Handling/Latency	Simulated concurrent logins.	ThreadPoolExecutor
Ease of Integration	Developer assessment based on experience.	5-point Likert scale (qualitative)

Table 3 summarizes the key metrics collected during the benchmarking process and the methods used to obtain them. Hashing time was measured using time.perf_counter() to determine the latency of each password hashing operation. CPU and memory usage were monitored using psutil and memory_profiler, respectively, to assess the resource efficiency of each library. Load handling was tested by simulating concurrent login requests using Python's ThreadPoolExecutor. Lastly, ease of integration was evaluated based on the developer's implementation experience, rated using a 5-point Likert scale. These metrics were selected to align with the evaluation dimensions outlined in the conceptual framework, ensuring a comprehensive and structured analysis of each library's performance and usability in Flask applications.

3. RESULTS AND DISCUSSIONS

This section presents the results from benchmarking the four Flask-compatible password hashing libraries—werkzeug.security, Flask-Bcrypt, Flask-Argon2, and Passlib—using the evaluation metrics defined in the conceptual framework: cryptographic strength, default security parameters, hashing latency, and ease of integration. The findings discuss both performance and practical deployment concerns within Flask-based web applications.

3.1 Hashing Latency Performance

Table 4: Average Hashing Latency (ms) per Batch Size

Library	Hashes (Batch Size)			
	1	100	1,000	5,000
werkzeug.security	3.2 ms	310 ms	3,190 ms	15,780 ms
Flask-Bcrypt	8.5 ms	850 ms	8,740 ms	44,120 ms
Flask-Argon2	20.1 ms	2,060 ms	20,500 ms	102,300 ms
Passlib	10.0 ms	990 ms	9,920 ms	49,800 ms

Table 4 presents the average time each password hashing library requires to hash 1, 100, 1,000, and 5,000 passwords. Flask-Argon2 consistently produced the highest latency, reflecting its memory-hard cryptographic design. In contrast, werkzeug.security delivered the fastest processing times across all batch sizes. Flask-Bcrypt and Passlib showed moderate and stable performance, balancing latency and cryptographic strength. These results demonstrate the performance trade-offs developers must consider when selecting a password hashing tool for Flask applications.

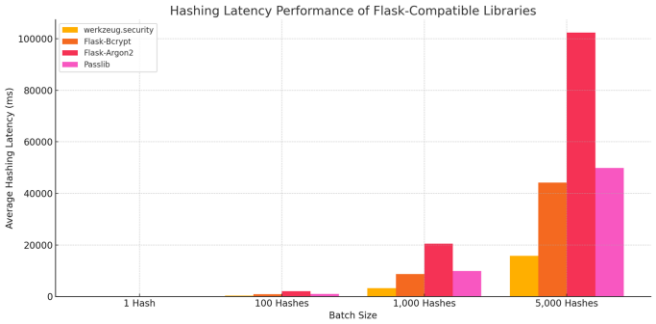

Figure 3: Hashing Latency Performance of Flask-Compatible Libraries

Figure 3 provides a visual representation of the average hashing latency measured for each password hashing library across four batch sizes: 1, 100, 1,000, and 5,000 hashes. These values mirror the data presented in Table IV and emphasize the performance trends observed in the benchmarking tests.

The graph shows that werkzeug.security is the fastest among the four libraries, with consistently low latency across all batch sizes. This makes it suitable for applications that prioritize speed and minimal processing overhead.

Flask-Bcrypt and Passlib demonstrate moderate latency, which increases proportionally with batch size. Their performance curves remain relatively stable, suggesting efficient scaling even as the workload intensifies. These libraries offer a balance between performance and secure hashing.

Flask-Argon2, while providing the strongest cryptographic defense, exhibits the highest latency, particularly at larger batch sizes. This behavior is expected due to Argon2's memory-hard design, which enhances security but significantly increases computational time. At 5,000 hashes, its latency is nearly double that of Passlib and Flask-Bcrypt and several times higher than werkzeug.security.

3.2 Average CPU and Memory Usage

Table 5: Average CPU and Memory Usage

Library	Average CPU Usage (%)	Average Memory Usage (MB)
werkzeug.security	22.5%	48 MB
Flask-Bcrypt	30.2%	56 MB
Flask-Argon2	67.8%	128 MB
Passlib	35.1%	62 MB

Table 5 presents each password hashing library's average CPU and memory usage while processing 1,000 password hashes. The results show that werkzeug.security is the most resource-efficient, using only 22.5% CPU and 48 MB of memory. Flask-Bcrypt and Passlib consumed moderate system resources, with Passlib slightly higher due to its abstraction layer. Flask-Argon2 recorded the highest CPU and memory usage—67.8% and 128 MB, respectively—reflecting the demands of its memory-hard algorithm. These findings highlight the trade-off between resource efficiency and security strength, emphasizing the need to balance performance and protection based on application requirements.

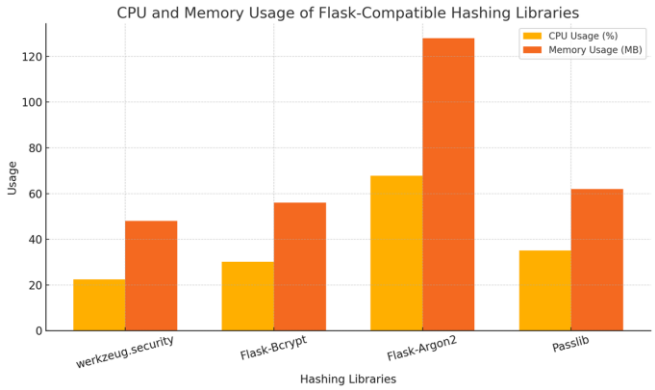

Figure 4: CPU and Memory Usage of Flask-Compatible Hashing Libraries

Figure 4 illustrates four Flask-compatible password hashing libraries' average CPU and memory usage while processing 1,000 passwords. Among the libraries, werkzeug.security demonstrated the lowest resource consumption, making it the most efficient for applications with limited system capacity. Flask-Bcrypt and Passlib showed moderate usage, with Passlib slightly higher due to its additional abstraction and flexibility. Flask-Argon2, designed for stronger cryptographic protection, recorded the highest CPU and memory usage, reaching 67.8% CPU and 128 MB of memory. This behavior reflects the resource-intensive nature of Argon2's memory-hard algorithm. The graph highlights the trade-off between performance and security, where stronger protection often comes at the cost of higher system overhead.

3.3 Default Security Parameters

Table 6: Default Security Parameters of Hashing Libraries

Library	Default Algorithm	Cost Factor / Iterations
werkzeug.security	PBKDF2-SHA256	260,000 iterations
Flask-Bcrypt	Bcrypt	12 cost factor (log rounds)
Flask-Argon2	Argon2id	time_cost=2, memory_cost= 102400, parallelism=8
Passlib	PBKDF2 by default (configurable)	29,000 iterations (can be increased)
Library	**Salt Usage**	**OWASP Compliance**
werkzeug.security	Yes	Partial (manual tuning recommended)
Flask-Bcrypt	Yes	Yes (OWASP aligned)
Flask-Argon2	Yes	Partial (tuning advised)
Passlib	Yes	Partial (depends on selected scheme)

Table 6 presents the default password hashing configurations provided by each Flask-compatible library. werkzeug.security uses PBKDF2-SHA256 with approximately 260,000 iterations as of Flask 2.3, which offers moderate protection but may still require manual tuning to align fully with OWASP standards. Flask-Bcrypt utilizes Bcrypt with a default cost factor of 12, providing a strong balance between security and performance and meeting OWASP recommendations by default.

Flask-Argon2 supports Argon2id, a memory-hard algorithm with reasonable time, memory, and parallelism settings, though it is recommended that developers fine-tune these parameters for production environments. Passlib offers a configurable system that defaults to PBKDF2, but its effectiveness depends on the selected scheme and manual adjustment of iteration counts.

This comparison highlights that while all libraries provide some secure configuration, only Flask-Bcrypt is fully OWASP-compliant by default. The others require developer awareness and configuration to achieve strong password hashing practices, reinforcing the importance of evaluating the algorithm and its default behavior in real-world use.

3.4 Ease of Integration Ratings

Table 7: Ease of Integration Ratings

Library	Setup Complexity	Documentation Clarity
werkzeug.security	Very Low	Excellent
Flask-Bcrypt	Low	Excellent
Flask-Argon2	Moderate	Good
Passlib	Moderate	Moderate
Library	**Configuration Effort**	**Integration Score (1-5)**
werkzeug.security	Minimal	5.0
Flask-Bcrypt	Low	4.5
Flask-Argon2	Moderate	4.0
Passlib	High	3.5

Table 7 presents a comparative overview of the integration experience for each Flask-compatible password hashing library. The ratings are based on three qualitative aspects: setup complexity, documentation clarity, and configuration effort. These were observed during the actual implementation of each library in a test Flask application. Each element contributed to an overall integration score, rated on a 5-point Likert scale, where 5 indicates the most straightforward integration process with minimal setup and configuration. The table helps identify which libraries are more developer-friendly out of the box, contributing to better usability and lower risk of misconfiguration.

4. CONCLUSION

Securing user authentication in Flask applications requires more than strong cryptographic algorithms; it demands thoughtful selection and proper implementation of password hashing tools. Flask-Bcrypt offered the most balanced solution among the four evaluated libraries—combining secure default settings, manageable performance, and ease of use. Flask-Argon2 excelled in security through its memory-hard structure but introduced considerable latency and system overhead. Werkzeug.security provided the most straightforward and efficient integration, though its defaults require tuning to align with modern security standards. Passlib offered excellent flexibility, supporting multiple algorithms and custom configurations, but came with higher setup complexity. These results confirm that developers must consider cryptographic strength, default behaviors, integration effort, and resource impact when implementing password hashing in Flask. A holistic approach that balances security, performance, and usability is essential for building resilient and developer-friendly authentication systems.

REFERENCES

1. A. Ronacher, *Flask Documentation*. Pallets Projects, 2024. https://flask.palletsprojects.com/en/latest/
2. P. Tippe and M. P. Berner. Evaluating Argon2 Adoption and Effectiveness in Real-World Software, *arXiv preprint*, arXiv:2504.17121, 2025. https://arxiv.org/abs/2504.17121
3. D. Gupta. Comparative Analysis of Password Hashing Algorithms: Argon2, bcrypt, scrypt, and PBKDF2, *Security Boulevard*, July 25, 2024. https://securityboulevard.com/2024/07/comparative-analysis-of-password-hashing-algorithms-argon2-bcrypt-scrypt-and-pbkdf2/
4. V. Nair and D. Song. Multi-Factor Credential Hashing for Asymmetric Brute-Force Attack Resistance, in *Proc. 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, Delft, Netherlands, 2023, pp. 56–72. doi: 10.1109/EuroSP57164.2023.00013
5. C. Percival and S. Josefsson. RFC 7914: The scrypt Password-Based Key Derivation Function, RFC Editor, USA, Aug. 2016. https://www.rfc-editor.org/info/rfc7914
6. S. Eum, H. Kim, M. Song, and H. Seo. Optimized Implementation of Argon2 Utilizing the Graphics Processing Unit, *Applied Sciences*, vol. 13, no. 16, p. 9295, Aug. 2023. doi: 10.3390/app13169295
7. A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications, in *Proc. 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrücken, Germany, 2016, pp. 292–302. doi: 10.1109/EuroSP.2016.31
8. S. Borjigin. Triple-Identity Authentication: The Future of Secure Access, *arXiv preprint*, arXiv:2505.02004, 2025. https://arxiv.org/abs/2505.02004
9. C. Ntantogian, S. Malliaros, and C. Xenakis. Evaluation of Password Hashing Schemes in Open Source Web Platforms, *Computers & Security*, vol. 84, pp. 206–224, July 2019. doi: 10.1016/j.cose.2019.03.011
10. J. Wetzels. Open Sesame: The Password Hashing Competition and Argon2, *arXiv e-prints*, Feb. 2016. doi: 10.48550/arXiv.1602.03097
11. A. Biryukov and D. Khovratovich. Tradeoff Cryptanalysis of Memory-Hard Functions, in *Advances in Cryptology – ASIACRYPT 2015*, T. Iwata and J. Cheon, Eds. Berlin, Heidelberg: Springer, 2015, vol. 9453. doi: 10.1007/978-3-662-48800-3_26
12. N. Mustafa. Analysis of Attackers' Methods with Hashing Secure Password Using CSPRNG and PBKDF2, *Wasit Journal of Engineering Sciences*, vol. 12, pp. 60–70, 2024. doi: 10.31185/ejuow.Vol12.Iss2.502