Performance Analysis And Parameter Optimization Of Argon2id For Secure Password Hashing

G Venu Gopal[1]
Assistant Professor, Department of Computer Science and Engineering
Vasireddy Venkatadri Institute of Technology
Nambur, Guntur District, Andhra Pradesh, India

Julakanti Thanmai Krishna[2] **Gudavalli Kumar**[3] **Jitta Nikhita**[4] **Kamunuri Kalyan Kumar**[5]
Undergraduate Students, Department of Computer Science and Engineering
Vasireddy Venkatadri Institute of Technology
Nambur, Guntur District, Andhra Pradesh, India

Abstract—**The secure storage of user credentials is a critical requirement for modern authentication systems. This project focuses on analyzing and implementing Argon2id, the recommended variant of the Password Hashing Competition winner, to evaluate its efficacy in secure password storage [2]. Argon2id provides robust resistance against brute-force and GPU-based cracking attacks through its configurable parameters, specifically time cost, memory cost, and parallelism [3]. The main goal of this research is to evaluate how parameter tuning affects both performance and security in real-world environments and to perform a systematic comparative evaluation of Argon2id and established password hashing algorithms such as bcrypt, scrypt, and PBKDF2.**

To accomplish this, a prototype login and registration system was implemented using Python and Flask, incorporating an automated benchmarking module to measure execution time and memory usage, and overall efficiency [1], [6], [10]. The comparative analysis and performance benchmarking highlight the trade-offs between usability and security under varying configurations [5], [7]. The results demonstrate that Argon2id effectively balances speed and memory hardness, providing strong resistance against brute-force attacks while maintaining practical authentication latencies. Ultimately, this paper provides practical recommendations and guidelines for developers to adopt secure parameter choices in CPU-based web environments while considering real-world system constraints [8].

I. Introduction

In the digital landscape, passwords remain the primary mechanism for user authentication across information technology systems. The safe storage of these credentials is of prime importance for the integrity of the system. This is because the storage of these passwords in plaintext form is catastrophic, as the exposure of one database compromises all users' credentials. To avoid this problem, contemporary systems of authentication use hashes of the passwords rather than the plaintext.

In the past, general cryptographic hash functions such as MD5 and SHA-1 have been used for this purpose. However, these traditional algorithms were intrinsically designed for computational speed and efficiency. While beneficial for data integrity checks, this rapid execution renders them highly susceptible to brute-force and dictionary attacks, particularly with the advent of highly parallel Graphics Processing Units (GPUs). Attackers leveraging GPU acceleration can compute billions of hash guesses per second, effectively neutralizing the security provided by fast hash functions. Consequently, there is a critical need for password hashing algorithms that are intentionally computationally expensive and memory-hard.

To counter advanced cracking techniques, modern password hashing schemes are specifically engineered to require significant memory and processing time. Among the prominent algorithms-including PBKDF2, bcrypt, and scrypt-Argon2 emerged as the winner of the Password Hashing Competition (PHC) and is regarded as a robust, state-of-the-art algorithm [9], [12]. The project focuses on analyzing and implementing Argon2id, the recommended variant of the Password Hashing Competition winner, for secure password storage in authentication systems [2]. Argon2id provides a secure mechanism against cracking attacks by combining data-dependent and data-independent memory access patterns, offering resistance against brute-force and GPU-based cracking attacks through its configurable parameters such as time cost, memory cost, and parallelism [3], [11].

The primary objective of this project is to evaluate how parameter tuning affects both performance and security in real-world environments, and to compare Argon2id with legacy password hashing schemes such as bcrypt, scrypt, and PBKDF2 [4]. By systematically analyzing these configurable parameters, this research highlights the trade-offs between usability and security, and provides practical guidelines for developers to select secure parameter choices for CPU-based environments under real-world constraints [7], [8].

II. Background and Related Work

In order to understand the context in which the performance analysis of Argon2id was conducted, it is necessary to grasp the basic concepts of password hashing algorithms and the optimization of such algorithms.

A. Related Work

Over the years, there has been a lot of literature on the computational limits of these password hashing algorithms. In particular, Eum et al. [1] published a study on the optimization of Argon2 using GPU devices. According to this study, it was observed that existing GPU-based implementations of pass- word hashing algorithms faced a critical bottleneck because of the huge data transfer overhead involved in transferring 1024 bytes of initialized memory blocks from the CPU to the GPU. To resolve this, Eum et al. proposed a hybrid implementation where the secondary expansion phase of initialization is offloaded directly to the GPU, utilizing high-speed shared memory for Blake2b computations. Their optimized method significantly reduced CPU-to-GPU data transfer delays and demonstrated a 5-to-6 fold speedup in processing large batches of passwords compared to CPU-only execution. While Eum et al. focused on optimizing attacker-side parallel brute-forcing models and low-level kernel improvements, our current project builds upon this foundation by evaluating how these inherent algorithmic costs affect legitimate, defensive web-server implementations.

B. Cryptography Basics and Memory-Hardness

While encryption is a two-way function intended to hide information, then reveal it, using a key, cryptographic hash- ing is a one-way function. ”A password hash function is a deterministic one-way mathematical function that transforms a password of arbitrary length in plaintext form into a fixed- length ’digest’ or ’bit array.’”

The key properties of a good password hash function are pre-image resistance, where a password hash function should be impossible to reverse; second pre-image resistance, where a password hash function should not be able to produce a different input that produces an identical output to a known input; and collision resistance, where a password hash function should not be able to produce two different inputs that produce an identical output.

To combat precomputed tables, known as dictionaries or rainbow tables, modern password schemes use a salt: a random sequence of characters, which is combined with the password before hashing. While salting prevents hash collisions for identical passwords, salting does not slow down hashing in any way.Consequently, modern password hashing schemes deploy “memory-hard” functions. These algorithms require a significant, tunable amount of Random Access Memory (RAM) to compute the hash, thereby neutralizing the paral- lel processing advantages of Application-Specific Integrated Circuits (ASICs) and GPUs used by attackers [2].

C. Legacy Hashing Algorithms

Prior to the standardization of Argon2, the industry relied on several key algorithms to secure credentials. In our comparative benchmarking, we evaluate the following legacy schemes:

- **PBKDF2 (Password-Based Key Derivation Function 2):** This is a password-based key derivation function that has a key-stretching algorithm, and just like the pseudorandom function and HMAC-SHA256, this function is computed in a loop, making it vulnerable to GPU attack because it is slow and requires minimal memory.
- **bcrypt:** The scheme uses key stretching with Blowfish encryption and has a cost factor and built-in salting. It uses more memory than PBKDF2 but still lacks in coping with hardware attacks.
- **scrypt:** The scheme was specifically designed to be memory-hard, meaning it produces large quantities of pseudorandom data and accesses it in a pseudo-random manner. It was designed to make attacks with expensive hardware costly.

D. Argon2 and its Variants

In response to the limitations and weaknesses of existing password hashing algorithms in defense against ASIC and GPU attacks, the international cryptography community initiated the Password Hashing Competition, or PHC, in 2013, and Argon2 was declared the winner in 2015 [9]. The Argon2 family is based on a customizable architecture that leverages the Blake2b cryptographic hash function, and there are three distinct variants in the Argon2 family, characterized by the following:

1) **Argon2d:** It maximizes data dependency in memory access, making it highly resistant to GPU attacks, although it is vulnerable to side-channel timing attacks.
2) **Argon2i:** It uses data-independent memory access, and its design is optimized for side-channel attack defense, sacrificing a bit on its ability to resist time-memory trade-off attacks.
3) **Argon2id:** It is a hybrid that combines the advantages of Argon2i and Argon2d, where Argon2i is employed in the first half of the first iteration, and Argon2d is employed in subsequent iterations, and Argon2id is now recommended as the standard for web authentication [3].

E. Argon2 Architecture

The internal execution of the Argon2 algorithm consists of three primary stages:

1) **Initialization:** The algorithm applies the user’s password in plain text, salt value, and various configuration values (time, memory, and parallel) using the Blake2b function to produce an initial hash block

of size 64 bytes (H_0).

2) **Block Filling:** Argon2 allocates a memory matrix divided into defined "lanes" (threads) and iteratively fills blocks of memory. Each block is generated based on a combination of previously computed blocks, strictly enforcing the memory-hard requirement.

3) **Final Block Creation and Digest:** Once all values are stored in the memory matrix for a specified number of iterations (time cost), a final password digest is created by hashing all final blocks from all lanes.

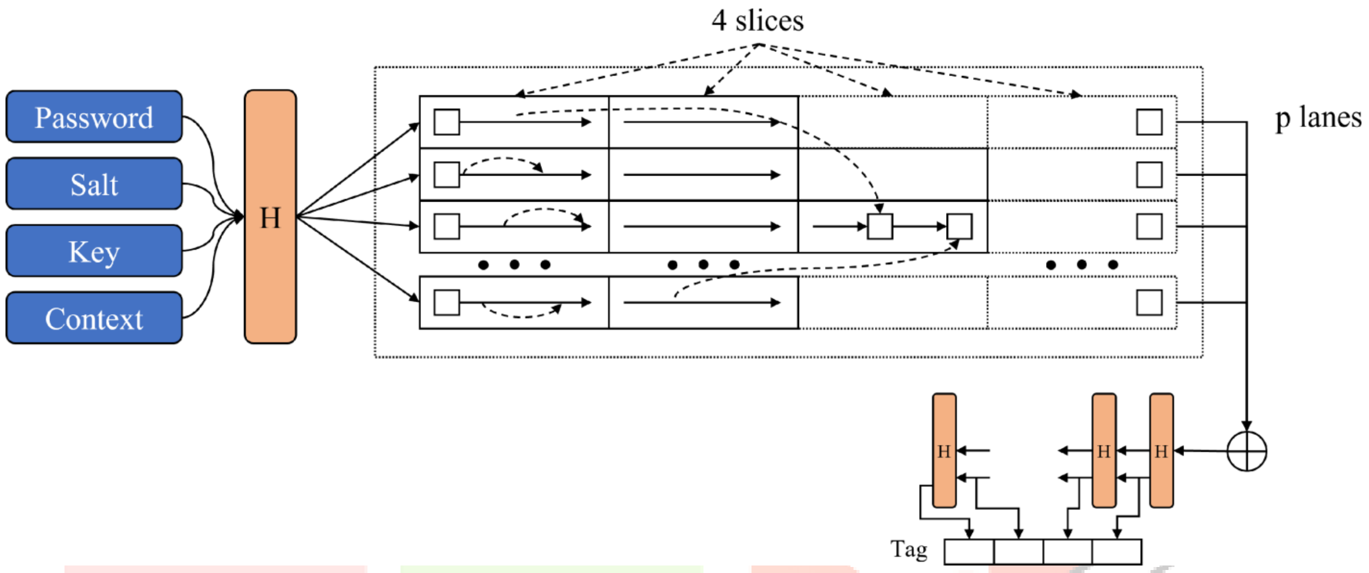

Fig. 1. The internal architecture and block-filling process of the Argon2 algorithm.

III. System Architecture and Methodology

To practically evaluate the performance and security trade-offs of the Argon2id algorithm, we implemented a complete, end-to-end full-stack web application. The system architecture is divided into two primary components: a functional authentication prototype simulating real-world user traffic, and an automated benchmarking module designed for systematic algorithm evaluation.

A. Authentication System Prototype

The authentication prototype was developed to model a standard, modern web server environment. It is implemented with the help of Python 3.11. In addition to that, it uses the Flask framework to develop a RESTful API.

As far as the storage of the data is concerned, it makes use of MongoDB for the persistent storage of the data in the system. It has one collection named `users`, which is used for the storage of the user credentials in the system. Unlike the storage of the password in the form of plain text, it stores the password in the form of hash using the `argon2-cffi` library. A base configuration is used to generate a hash value for passwords when users are registered in the system. For instance, a time cost of 2 is applied, a memory size of 65,536 KB is applied, and a parallelism degree of 2 is applied. However, the hash string, which inherently contains salt values and parameters used during password hashing, is stored in a safe manner in the database. When a user is logged in, the hash string is retrieved in a safe manner. Hence, password validation is achieved in a safe manner. Additionally, simulation of the delay in password validation is applicable for a user.

B. Automated Benchmarking Module

The core analytical component of this research is the automated benchmarking suite, engineered to systematically test various hashing configurations and compare Argon2id against legacy algorithms (bcrypt, scrypt, and PBKDF2).

The benchmarking module executes via an asynchronous API endpoint that accepts a matrix of target parameters. For Argon2id, the module iterates through defined sets of time costs (t), memory costs (m), and parallelism degrees (p). For each unique configuration, the module performs the following methodology to ensure accurate measurement:

1) **Workload Generation:** A test password of a fixed length is iteratively hashed a number of times to mimic the workload and determine the average metrics.
2) **Precision Timing:** The latency of execution of the hash function is determined by utilizing Python's high-resolution `time.perf_counter()` to record the exact time taken during the hash generation.

3) **Resource Monitoring:** The usage of resources such as CPU and memory can be monitored using the `psutil` library. The CPU usage will be monitored in terms of percentage during the execution of the hash function. The memory overhead will be monitored by calculating the difference in RSS values before and after the execution of the hash function.
4) **Data Aggregation:** The `bcrypt` library will be used for legacy comparison, while the `hashlib` library of Python will be used for the scrypt algorithm along with PBKDF2 using `pbkdf2_hmac` and SHA-256.

Upon completion of the testing cycles, the benchmarking module will aggregate the performance results, including total execution time, average execution time per hash, CPU utilization percentage, and memory utilization, which will be committed to the `benchmarks` MongoDB collection. The system will also export the data in a standardized CSV format, which can be used for data analysis, including interactive visualizations such as heatmaps and performance charts, using Matplotlib, Seaborn, and Plotly.

The proposed approach guarantees that the performance of the Argon2id password hash function will be assessed relative to real-world, empirically measurable server metrics rather than theoretical limits.

IV. Experimental Setup

To conduct a comprehensive evaluation of Argon2id's performance and parameter optimization, we established a structured experimental matrix. The experiments were designed to measure the impact of varying cryptographic parameters on system resources and execution latency, and to establish a baseline using legacy algorithms.

A. Tunable Parameters of Argon2id

Argon2id algorithm has various tunable parameters, which can be managed by the system administrator depending on the requirements related to the trade-off between performance and security against brute-force attacks Our benchmarking suite isolated and manipulated the following specific variables:

- **Memory Cost (m):** This parameter is used to specify the memory that is required during the hashing process. Increasing this parameter raises the financial and computational cost for attackers utilizing parallel GPU or ASIC hardware. In our tests, we used memory costs of 16,384 KB, 65,536 KB, and 262,144 KB. These values correspond to memory sizes of 16 MB, 64 MB, and 256 MB.

- **Time Cost (t):** Determines the number of computational iterations (passes) are carried out on the memory matrix. Higher values linearly increase the computation time. Our tests evaluated time costs of 1, 2, and 4 iterations.

- **Parallelism (p):** Controls the number of independent threads (or lanes) that are employed to perform the hash computation. Although this maximizes the efficiency of legitimate multi-core CPUs, it also controls the level of parallelization that the attacker can utilize. Our experiments tested parallelism degrees of 1, 2, and 4.

- **Hash Length:** The length of the final output hash string. For consistency across all Argon2id tests, this was held constant at 32 bytes, accompanied by a 16-byte cryptographic salt.

B. Comparative Baselines

To evaluate how Argon2id compares to other hashing algorithms, we implemented other popular hashing algorithms as comparative baselines. The parameters used for the comparative baselines were fixed and set to recommended values for a typical legacy web server:

- **bcrypt:** Configured with a work factor (rounds) of 12.
- **scrypt:** Configured with parameters $N = 16384$ (CPU/Memory cost), $r = 8$ (block size), and $p = 1$ (parallelization), producing a 32-byte derived key.
- **PBKDF2:** Configured to utilize the HMAC-SHA256 pseudorandom function with 100,000 iterations.

C. Hardware and Execution Environment

To ensure reproducibility and accurately simulate a standard backend web environment, all benchmarks were executed within a controlled software stack. The benchmarking module was implemented in Python 3.11. System-level resource monitoring was handled via the Python `psutil` library, capturing CPU utilization and Resident Set Size (RSS) memory differentials.

To eliminate variance from background OS processes, each unique parameter combination was iterated 5 times. The reported metrics in our results represent the arithmetic mean of these iterations, ensuring statistical stability in the recorded execution times and memory footprints.

V. Results and Performance Analysis

The benchmarking module performed a series of hashing operations across the defined parameter grid. The results provide quantitative evidence of configurable memory-hardness of Argon2id and its comparative performance against legacy algorithms. All reported execution times represent the average time per hash (in milliseconds) derived from batches of 5 consecutive iterations to ensure statistical stability.

A. Impact of Memory Cost (m)

The core defensive mechanism of Argon2id is its memory-hardness. Our results confirm that scaling the memory requirement imposes a severe and intentional computational delay . Keeping the time cost and parallelism constant at their minimums ($t = 1$, $p = 1$), configuring the algorithm to allocate 16,384 KB (16 MB) of memory resulted in a highly efficient average hashing time of 38.94 ms . However, increasing the memory cost to 262,144 KB (256 MB) increased the average latency to 700.10 ms . This substantial increase demonstrates how effectively Argon2id can be tuned to bottleneck attacker hardware, which typically possesses limited high-speed RAM per processing core.

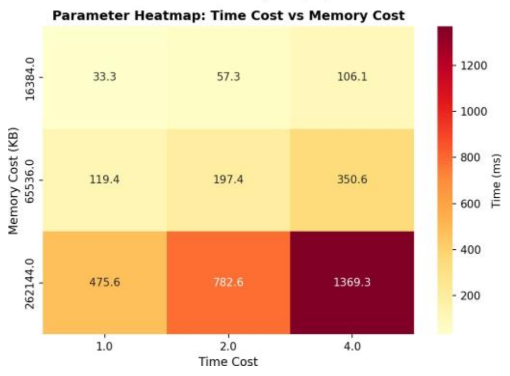

Fig. 2. Heatmap illustrating the combined effect of time cost and memory cost parameters on Argon2id hash execution time.

B. Impact of Time Cost (t)

The time cost parameter determines the number of iterations the algorithm performs on the allocated memory matrix. The benchmark data illustrates a nearly linear correlation between time cost and execution latency . Observing a medium memory configuration ($m = 65, 536$ KB) with a single thread ($p = 1$), the average hashing times scaled predictably: 172.44 ms for $t = 1$, 265.78 ms for $t = 2$, and 499.37 ms for $t = 4$. This allows administrators to increase the computational burden on attackers without linearly increasing the memory footprint of the server.

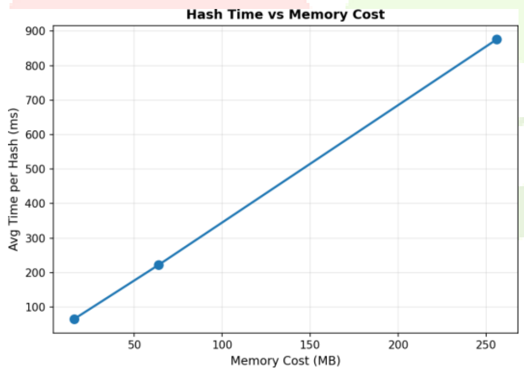

Fig. 3. Impact of Argon2id memory cost on average hash execution time.

TABLE I
Impact of Memory Cost on Hash Computation Time

Memory Cost (MB)	Avg Time per Hash (ms)
32	65
64	220
256	870

C. Impact of Parallelism (p)

Parallelism allows the algorithm to split the hashing workload across multiple CPU threads. The data demonstrates that maximizing thread utilization significantly reduces execution time for legitimate users on multi-core servers . Under the most resource-intensive tested configuration ($t = 4$, $m = 262, 144$ KB), executing the hash on a single thread ($p = 1$) resulted in an average time of 2,096.30 ms . When the workload was distributed across four threads ($p = 4$), the latency dropped by nearly 64% to 753.71 ms . This proves that highly secure, memory-intensive configurations can be practically deployed if the host server leverages concurrent processing.

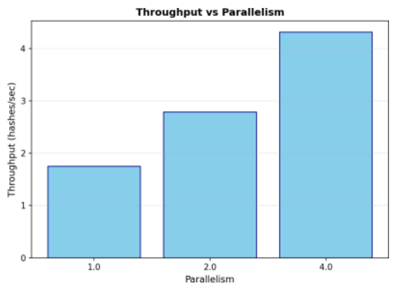

Fig. 4. Effect of parallelism on Argon2id hashing throughput.

TABLE II
Throughput as a Function of Parallelism

Parallelism	Throughput (hashes/sec)
1	1.7
2	2.8
4	4.3

D. Comparative Algorithm Analysis

To contextualize Argon2id's performance, we benchmarked it against standard implementations of bcrypt, scrypt, and PBKDF2 . Under typical legacy configurations, PBKDF2 and scrypt executed the fastest, averaging 112.08 ms and 121.30 ms per hash, respectively . The bcrypt algorithm, operating with a work factor of 12, averaged 768.05 ms per hash . By tuning its parameters, Argon2id can be explic-

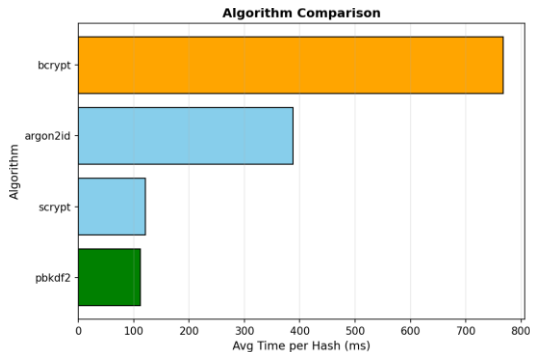

Fig. 5. Comparison of average per-hash execution time for common password hashing schemes, including PBKDF2, scrypt, Argon2id, and bcrypt.

itly configured to mimic or exceed these baseline latencies while offering superior architectural security. For instance,

TABLE III
AVERAGE TIME PER HASH FOR DIFFERENT ALGORITHMS

Algorithm	Avg Time per Hash (ms)
PBKDF2	110
scrypt	120
Argon2id	390
bcrypt	760

an Argon2id configuration of $t = 4, m = 16, 384$ KB, and $p = 2$ averaged 107.04 ms , rivaling the speed of PBKDF2 while guaranteeing memory-hardness. Conversely, raising the parameters to $t = 4, m = 262, 144$ KB, and $p = 4$ achieves a latency of 753.71 ms , closely matching bcrypt's delay but with exponentially higher resistance to GPU cracking due to the massive 256 MB memory requirement per hash.

VI. Discussion and Practical Recommendations

The observed results of this study shows that Argon2id is a highly configurable algorithm. While conventional cryptographic algorithms can only rely on the brute force of computational iterations (key stretching), the memory-hardness of Argon2id's design allows for a multi-dimensional defense strategy. However, the flexibility of the Argon2id design also creates a significant engineering challenge in finding the right balance of security and usability.

A. The Security vs. Usability Trade-off

The primary objective of a password hashing algorithm is to maximize the financial and computational cost for an attacker attempting to crack hashes offline. As demonstrated in our results, increasing the memory cost (m) and time cost (t) significantly achieves this goal. However, these identical costs are incurred by the legitimate authentication server during every user login attempt.

If the parameters are too high, e.g., $t = 4$, $m = 262, 144$ KB, $p = 1$, or higher than 2 seconds of latency, it will be very secure but could irritate legitimate users and invite Denial of Service (DoS) attacks, whereby an attacker attempts to overwhelm the server's CPU and RAM by flooding the login endpoint. If the parameters are too low, e.g., latency ¡ 50 ms, it does not effectively utilize its memory hardness and is vulnerable to GPU-based attacks.

B. Calculating the Optimal Configuration

In order to solve this trade-off, it is necessary to establish a budget for the authentication latency. The standard for the hashing latency is between 250 and 500 milliseconds, according to the Open World Wide Web Application Security Project (OWASP) and RFC 9106 [3]. This is completely imperceptible to a human user, while it is computationally infeasible for a brute-force attacker.

Based on our benchmarking data, we propose the following practical recommendations for deploying Argon2id in CPU-based web environments:

1) **Maximize Memory First:** The most effective defense against GPU-based cracking is RAM exhaustion. The developer should also allocate the maximum memory available to his/her server infrastructure to service concurrent login requests. The recommended baseline should be 64 MB or $m = 65, 536$ KB, with scaling up to 256 MB or more for high-security requirements.
2) **Tune Execution Time:** Once the memory limit is established, adjust the time cost (t) to bring the total execution latency into the target 250–500 ms window. For example, we have shown in the test results that for a given value of $t = 2$ and $m = 65, 536$ KB on a single thread, the optimal value of latency is 265.78 ms.
3) **Leverage Parallelism:** If the authentication server has processors that support multiple cores, the value of the parallelism parameter, p, should be adjusted according to the number of cores that are available in the processor, which is usually 2 or 4 in most cases. This is to minimize the amount of time that the legitimate user has to wait in order to perform the hash computation, and at the same time, the attacker is forced to perform the guessing process with the same number of resources.
4) **Hardware-Specific Calibration:** Because hashing performance is strictly bound to the host machine's CPU architecture and RAM speed, developers must never rely blindly on default library parameters. The benchmarking module developed in this project should be executed directly on the production hardware prior to deployment to calibrate the exact t, m, and p values required to hit the target latency budget.

By adhering to these guidelines, system administrators can effectively implement Argon2id to future-proof their authentication systems against evolving hardware-accelerated attacks without degrading application performance.

VII. Limitations

Although this study successfully modeled the defensive performance of Argon2id on standard CPU-based server architectures, it has certain limitations. First, the benchmarking was conducted strictly within a CPU-bound environment.

VIII. Conclusion

One of the most critical aspects of modern authentication systems is secure storage of user credentials. With advancements in attacker hardware, such as highly parallelized GPUs and ASICs, cryptographic hash functions such as MD5, SHA256, and even bcrypt are becoming increasingly vulnerable.

This research systematically analyzed and evaluated Argon2id, the recommended variant of the Password Hashing Competition (PHC) winner, to quantify its effectiveness in real-world web environments. By developing an end-to-end authentication prototype and an automated benchmarking suite, we provide experimental evidence of the substantial impact of Argon2id's configurable parameters: memory cost (m), time cost (t), and parallelism (p).

As such, the findings of this study conclusively determine that Argon2id offers a far better trade-off between security and performance than traditional algorithms such as bcrypt, scrypt, and PBKDF2. This is due to the ability of administrators to deliberately slow down attackers' hardware by greatly increasing the memory cost factor, while at the same time not significantly impacting the response times of legitimate users, particularly when taking advantage of parallel processing capabilities of multiple cores available on most modern computer hardware. This study also developed guidelines for developers, which highlighted the importance of parameter calibration based on host hardware for optimal 250-500 ms latency budgeting.

IX. Future Work

Future work should involve extensive, data-driven GPU-cracking simulations to quantify exactly how much attacker time and financial cost is added per megabyte of increased memory cost.

Additionally, future research could explore multi-GPU scaling impacts on Argon2id's resistance, the integration of secret cryptographic "peppers" stored in external hardware security modules (HSMs), and the implementation of dynamic rate-limiting algorithms to further harden web-facing authentication endpoints against distributed brute-force attacks.

References

[1] 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, 2023.

[2] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *Proc. IEEE EuroS&P*, 2016, pp. 292–302.

[3] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications," RFC 9106, Internet Engineering Task Force (IETF), Aug. 2021.

[4] D. P. Provos and M. Mazieres, "A future-adaptable password scheme," in *Proc. USENIX Annual Technical Conference*, 1999.

[5] I. Alkhwaja et al., "Password cracking with brute force algorithm and dictionary attack using parallel programming," *Applied Sciences*, vol. 13, no. 10, 2023.

[6] C. H. Lin et al., "On the performance of cracking hash function SHA-1 using cloud and GPU computing," *Wireless Personal Communications*, vol. 109, pp. 491–504, 2019.

[7] A. Biryukov and D. Khovratovich, "Tradeoff cryptanalysis of memory-hard functions," in *Proc. ASIACRYPT*, 2015.

[8] OWASP Foundation, "Password Storage Cheat Sheet," 2023. [Online].

[9] G. Hatzivasilis, I. Papaefstathiou, and C. Manifavas, "Password Hashing Competition-Survey and Benchmark," Cryptology ePrint Archive, Report 2015/265, 2015.

[10] Python Software Foundation, "Flask Documentation," https://flask.palletsprojects.com/

[11] A. Biryukov, "Design and analysis of memory-hard functions," Ph.D. dissertation, University of Luxembourg, 2016.

[12] NIST, "Digital Identity Guidelines," SP 800-63B, 2022.

[13] C. Percival, "Stronger Key Derivation Via Sequential Memory-Hard Functions," presented at *BSDCan*, Ottawa, Canada, 2009.

[14] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898, Internet Engineering Task Force (IETF), Sep. 2000.

[15] D. Boneh, C. Herder, and S. Ren, "Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks," in *Advances in Cryptology – ASIACRYPT*, 2016, pp. 220–248.

[16] J. Alwen and B. Blocki, "Efficiently Computing Data-Independent Memory-Hard Functions," in *Advances in Cryptology – CRYPTO*, 2016, pp. 241–271.

[17] W. Lee et al., "Efficient Optimization of PBKDF2-HMAC-SHA2 Password Cracking on GPUs," *IEEE Access*, 2024.

[18] N. Zheng et al., "An Empirical Study of Web Password Strength and Bounding Password Policies," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[19] J. Steube, "hashcat - advanced password recovery," 2024. [Online]. Available: https://hashcat.net/hashcat/

[20] Openwall, "John the Ripper password cracker," 2024. [Online]. Available: https://www.openwall.com/john/