A Comparative Analysis of PBKDF2 and Argon2id as Key Derivation Functions in SQLCipher

Aria Judhistira - 13523112

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: aria.judhistira2108@gmail.com, 13523112@std.stei.itb.ac.id

Abstract**—**SQLCipher is a widely adopted library for encrypting SQLite databases. It derives its AES-256 encryption key using PBKDF2 by default. While PBKDF2 is well-established and standardized, it lacks memory-hardness, making it vulnerable to highly parallelized GPU-accelerated attacks. Argon2id, the winner of the 2015 Password Hashing Competition, provides a more resilient alternative. However, its configurable memory requirements raise practical concerns regarding application latency in environments limited to resources. This paper provides a comparative analysis of PBKDF2-HMAC-SHA512 and Argon2id as key derivation functions within the operational context of SQLCipher, evaluating both algorithms based on raw derivation time, resistance towards parallel computation scaling, and database initialization overhead. Results show that PBKDF2 exhibits a linear scaling behavior between iteration count and latency, while Argon2id provides a multi-dimensional parameter space that achieves lower latency at comparable security configurations. The dictionary attack simulation demonstrates Argon2id's memory-hardness constrains parallel scaling more effectively than PBKDF2 whilst remaining within acceptable latency thresholds in operational context of SQLCipher. These findings support Argon2id as a suitable candidate for replacing PBKDF2 as the default key derivation function in SQLCipher deployments.

Keywords**—***SQLCipher, PBKDF2, Argon2id*

I. INTRODUCTION

The rise in mobile and desktop applications has made local data storage a focal attack point. The Open Worldwide Application Security Project Foundation, more commonly referred to as the OWASP Foundation, places it among its top 10 in mobile security risks as M9: Insecure Data Storage [1]. As a countermeasure, the library SQLCipher has become a widely adopted standard, providing 256-bit AES encryption for SQLite databases. At its core, SQLCipher utilizes a key derivation function, specifically PBKDF2 (Password-Based Key Derivation Function version 2), which derives a user's password into a cryptographically secure key that's suitable for AES encryption and decryption process [2].

Although PBKDF2 is a well-established algorithm for key derivation, it relies heavily on computational iterations to deter brute-force and dictionary attacks. This means that PBKDF2 is not truly memory-hard, which allows the usage of powerful parallel processing hardware such as Graphics Processing Units (GPUs) to significantly weaken its security [8]. Attackers then could utilize parallel architecture systems to compute high-throughput offline dictionary attacks which bypass the bottlenecks of a CPU. In response to this, there have been attempts to shift the paradigm of key derivation functions to memory-hard and future-proof algorithms. One of those algorithms, Argon2id, was selected as the ultimate winner of the 2015 Password Hashing Competition. Argon2id introduces configurable memory and parallelism costs alongside computational time, which forces attackers to use significant memory bandwidth, effectively removing the attacker's advantages of GPU computation and calculation acceleration. As a direct result, Argon2id is increasingly preferred as the new standard for password-derived keys in modern security standards.

Despite its effectiveness in resisting memory-intensive attacks whilst still being computationally expensive, Argon2id is not without faults. It consumes a substantial amount of memory and processing power by design, with overly aggressive configurations theoretically able to result in application delays. This is a concern, especially for mobile applications where processing capabilities and memory availability are considerably limited compared to desktop applications. This results in a practical paradox in Argon2id's usage in a database encryption engine such as SQLCipher, where its security and resistance towards attacks acts as a trade-off for user experience.

This paper presents a comparative study of PBKDF2 and Argon2id within the operational context of SQLCipher. It aims to compare these two algorithms through experiments involving key derivation time measurements, simulated dictionary attacks, and database overhead evaluations to further identify each algorithm's strengths and weaknesses. Furthermore, it seeks to decide whether Argon2id's perceived impracticality is acceptable as SQLCipher's key derivation function without exceeding user experience latency thresholds.

II. THEORETICAL FRAMEWORK

A. Hash Function

Hash functions serve as the basis for key derivation functions. According to [3], a hash function is defined as a function that receives an input of arbitrary size, compresses it, and outputs it as a fixed-size value, usually named as a hash value. A hash value is irreversible, making it distinct to an encryption-decryption algorithm. On the other hand, a

cryptographic hash function is a hash function that prioritizes security over computation speed and practicality. Known and generally used cryptographic hash functions include MD5, SHA-1, and SHA-256 [4]. A hash function is considered a cryptographic hash function if it satisfies three characteristics:

1. Collision resistant: it is difficult or infeasible to find two different inputs where both hash values are the same.
2. Preimage resistant: it is difficult or infeasible to find an input that produces a certain hash value.
3. Second preimage resistant: it is difficult or infeasible to find a second, different input that produces the same hash value as a given input.

B. Key Derivation Function

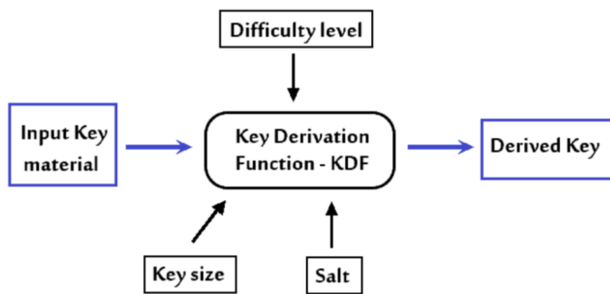

Fig. 1. Illustration of a key derivation function [5]

A key derivation function (KDF) is a cryptographic algorithm that generates a secure key from a single, secret initial value, usually a password. A KDF is commonly based on a cryptographic hash function, though that is not always the case as it could also rely on block ciphers or stream ciphers. As illustrated in Fig 1., a KDF can receive other parameters alongside said initial value, such as salt value, difficulty level, and key size. Generally, the salt value is appended to the initial value before it is hashed, with the difficulty level being the number of iterations of hashing it goes through before being output [5].

C. Password-Based Key Derivation Function Version 2

Password-Based Key Derivation Function version 2 (PBKDF2) is a KDF that was published by RSA Laboratories as part of Public-Key Cryptography Standard (PKCS) #5. PBKDF2 introduces CPU-intensive operations to slow down brute-force attacks against weak passwords. It achieves this by transforming said password into a cryptographically secure key through a repeated sequence of pseudorandom functions (PRFs), such as a hash function or HMAC [6].

According to RFC 2898 [7], PBKDF2 works as follows.

ALGORITHM 1: PBKDF2

Input:
Pseudorandom function ***PRF*** *with output length hLen*
Password ***P***
Salt ***S***
Iteration count ***c***
Intended length of derived key ***dkLen***
Output: *Derived key* ***DK***

1 ***hLen*** ← *output length of PRF in bytes*
2 ***if*** *dkLen > (2^32-1) * hLen* ***then stop***
3 ***l*** ← *ceil(dkLen / hLen)*
4 ***r*** ← *dkLen – (l – 1) * hLen*
5 ***initialize*** *DK as empty byte array*
6 ***F*** ← *function defined as exclusive or-sum of first c iterates of PRF*
7 ***for*** *i = 0* ***to*** *l* ***do***
8 | ***Ti*** ← *F(P, S, c, i)*
9 | ***Append*** *Ti to DK*
10 ***end for***
11 ***return*** *DK*

As discussed by [8], PBKDF2 utilizes a fixed, minimal amount of memory, making it highly susceptible to GPU attacks. This is caused by the lack of memory-hardness, where PBKDF2's design includes only a time-based security parameter through the number of iterations, but no parameter to increase memory usage. Besides that, [6] demonstrates that an attacker could bypass a significant portion of the computational effort required to derive the key by exploiting redundancies in standard HMAC-SHA-1 implementations. Because the user's password and the HMAC padding constants remain the same, an attacker could precompute the initial message blocks and reuse them, effectively skipping 50% of the required operations. Furthermore, attackers can omit useless XOR calculations within the SHA-1 algorithm by ignoring the strings of zeros used for message padding.

D. Argon2id

Argon2id is a variant of the Argon2 family, which won the 2015 Password Hashing Competition. Argon2's primary goal as a memory-hard function is to provide strong resistance towards brute-force attacks, especially against highly parallelized hardware like GPUs. Argon2id itself is a hybrid of two other variants of the Argon2 family: Argon2i and Argon2d. Argon2i uses data-independent memory access which prevents side-channel attacks, while Argon2d uses data-dependent memory access which maximizes resistance to GPU brute-force attacks and time-memory tradeoffs [9]. The combination of both strengths creates a middle-ground which covers up each other's weaknesses, leading to Argon2id working as Argon2i for the first half of the data pass and Argon2d for the rest [10].

According to RFC 9106 [10], Argon2id works as follows.

ALGORITHM 2: ARGON2ID

Input:
Password ***P***
Salt ***S***
Degree of parallelism ***p***
Desired output tag length ***T***
Memory size ***m***
Number of iterations ***t***
Optional secret key ***K***

```
        Optional associated data X
        Version number v
        Algorithm type y
    Output: Derived key DK

1   m' ← 4 * p * floor(m / 4p)
2   Buffer ← LE32(p) || LE32(T) || LE32(m) || LE32(t) ||
    LE32(v) || LE32(y) || LE32(length(P)) || P ||
    LE32(length(S)) || S || LE32(length(K)) || K ||
    LE32(length(X)) || X
3   H_0 ← BLAKE2b(Buffer, 64)
4   q ← m' / p
5   initialize matrix B of size p * q
6   for i = 0 to (p-1) do
7       B[i][0] ← VariableLengthHash(H_0 || LE32(0) ||
        LE32(i), 1024)
8       B[i][1] ← VariableLengthHash(H_0 || LE32(1) ||
        LE32(i), 1024)
9   end for
10  for pass = 0 to (t-1) do
11      for slice = 0 to 3 do
12          for i = 0 to (p-1) do
13              for j = 2 to (q-1) do
14                  NewBlock ← G(B[i][j-1], B[l][z])
15                  if pass = 0 then B[i][j] ← NewBlock
16                  else B[i][j] ← B[i][j] XOR
                    NewBlock
17              end for
18          end for
19      end for
20  end for
21  C ← B[0][q-1] XOR B[1][q-1] XOR ... XOR B[p-1][q-1]
22  DK ← VariableLengthHash(C, T)
23  return DK
```

The main tradeoff for Argon2id lies in its memory usage and necessary computational power required. As the number of each parameter grows (memory, iterations, and parallelism), the generated key would be much more secure in exchange for higher RAM usage, higher CPU usage, and the need for multi-core resource allocation [10].

E. SQLCipher

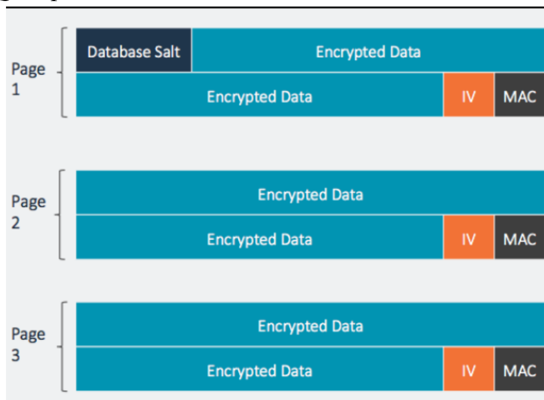

Fig. 2 Structure of SQLCipher-encrypted database pages [11]

SQLCipher is an open-source extension to the SQLite database library which provides transparent and on-the-fly encryption using AES-256-CBC. SQLCipher operates at the page level, where each database page is encrypted and decrypted individually. To further enhance security, it appends a 16-byte salt at the start of its first page, while a unique and random 16-byte initialization vector (IV) and a Message Authentication Code (MAC) for each page are appended to its end as shown in Fig. 2 [11], [12].

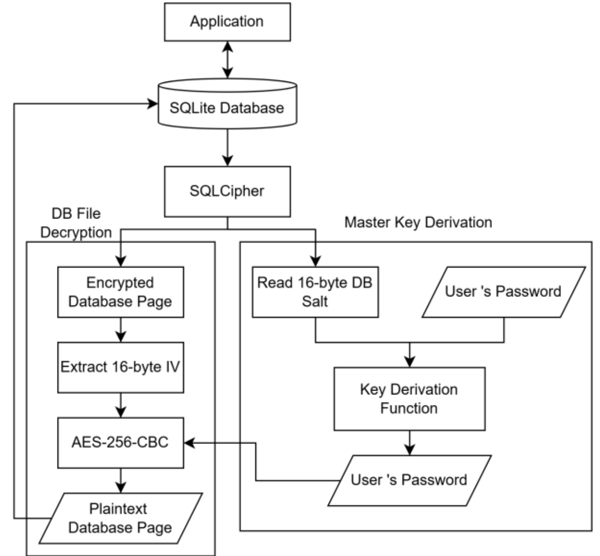

Fig. 3 SQLCipher master key derivation and decryption process

The encryption key used for the AES-256-CBC is derived from the password which the user provides. The password, alongside the salt appended to the first page of the database, is then passed through the designated KDF. By default, the KDF used is PBKDF2 paired with HMAC-SHA512 and executing 256,000 computational iterations. The extensive number of iterations produces a cryptographically secure 256-bit master key [12].

III. Methodology

To evaluate the performance and tradeoffs of PBKDF2 and Argon2id in the operational context of SQLCipher, there are three experiments designed. The experiments focused on raw derivation time, resistance to brute-force attacks, and practical impact on application startup latency.

A. Environmental Setup

All experiments were conducted locally in a consumer-grade computing environment equipped with a multi-core Intel Core i5 (11th Generation) processor. This CPU-bound environment reflects the hardware architecture of typical devices running SQLite and SQLCipher, while still providing the necessary cores to evaluate multi-thread parallelization.

The simulations themselves were implemented and run in Python 3, using the hashlib library for PBKDF2 operations, argon2-cffi library for Argon2id operations, the cryptography library to simulate AES-256-CBC database encryption

mechanics, and the cocurrent.futures module to enable multi-core parallel processes.

B. Key Derivation Function Benchmarking

The first experiment measured the raw execution time needed by each KDF to derive a 256-bit secret key from a static password and a 16-byte random salt within the user's operating environment. To account for background operating system processes noise and data stability, each configuration used in this experiment will be done five times, with the results being the average execution time. The configurations used in this experiment are the following:

- PBKDF2-HMAC-SHA512: Evaluated six number of iterations configuration being 64000, 100000, 210000, 256000, 500000, and one million iterations.
- Argon2id: Ten configurations were evaluated, all of which used a degree of parallelism of 1, with the configurations being as follows in Table 1.

TABLE 1. EXPERIMENT 1 ARGON2ID CONFIGURATIONS

Time Cost	Memory Size
1	16 MiB
1	19 MiB
2	19 MiB
3	12 MiB
1	46 MiB
2	46 MiB
1	64 MiB
2	64 MiB
3	64 MiB
2	128 MiB

C. Multi-Process Dictionary Attack Simulation

The second experiment aimed to quantify the resistance of both KDFs against offline dictionary attacks utilizing highly parallelized hardware. To demonstrate the vulnerability of the KDFs to parallel hardware, the methodology was designed to measure how efficiently each KDF scales across multiple processing cores.

A controlled dictionary attack was simulated using a 1,000-word subset of the Rockyou dataset. To ensure the worst-case scenario, the target password is excluded from the original dataset and appended to the end of the list. This forced each algorithm to evaluate every candidate, removing early-elimination bias.

The simulation was implemented using the Python concurrent.futures.ProcessPoolExecutor module to bypass the Global Interpreter Lock and spawn isolated worker processes. The attack was executed in two modes: a single-threaded execution, and a multi-process execution involving eight workers. The experiment measured the total system throughput in hashes evaluated per second with the following configurations:

- PBKDF2-HMAC-SHA512: 64000, 210000, 256000, and 1 million iterations.
- Argon2id: Evaluated three configurations, all of which used a degree of parallelism of 1. The three configurations are detailed in Table 2.

TABLE 2. EXPERIMENT 2 ARGON2ID CONFIGURATIONS

Time Cost	Memory Size
1	46 MiB
2	19 MiB
2	64 MiB

The primary metric derived from this simulation was parallel scaling efficiency, which was calculated as the speedup ratio between the multi-process and single-threaded throughput. Then, the measured throughputs were extrapolated to calculate the theoretical time required to traverse the whole Rockyou dataset, which has a total of 14.3 million entries.

D. Database Initialization Overhead

The final experiment aimed to evaluate the practical usability of KDF configurations by measuring application startup overhead. Because legacy Python SQLCipher wrappers lack support for Argon2id usage, the encryption pipeline was simulated using the native cryptography library instead as according to Fig. 3.

The experiment was divided into two phases. The first phase simulated the "Cold Start" sequence, which represents the total latency when unlocking a database. This phase timed the execution of a KDF to generate a 256-bit key, followed by the AES-256-CBC decryption of a 4096-byte database header page containing a 16-byte initialization vector. The second phase measured the query throughput by timing the bulk AES-256-CBC decryption of 500 database pages representing 2 MiB of data. This was done to demonstrate that the computational cost for deriving the key is only a one-time penalty that does not affect ongoing performances after the database was unlocked.

To ensure measurement stability, the cryptographic operations were executed in memory using randomly generated byte sequences matching the standard SQLite page size. The cold start sequence itself is executed five times, with the mean time being used as experiment result.

The configurations tested in this experiment involve four configurations of iterations amounting to 64000, 256000, 500000, and 1 million for PBKDF2. For Argon2id, the configurations are as follows in Table 3.

TABLE 3. EXPERIMENT 3 ARGON2ID CONFIGURATIONS

Time Cost	Memory Size
1	46 MiB
2	46 MiB
2	64 MiB
3	64 MiB

IV. RESULTS

This section presents the data collected across the three experiments devised for this research.

A. Key Derivation Function Benchmarking Results

This experiment measured the baseline execution time required by PBKDF2 and Argon2id to derive a 256-bit cryptographic key. Table 4. details the performance of

PBKDF2 across increasing iteration counts (*n*) and Argon2id across varying time (*t*) and memory (*m*) costs.

TABLE 4. KDF BENCHMARKING RESULTS

KDF	Configuration	Latency (ms)	± Std Dev (ms)
PBKDF2	*n=64000*	71.772	0.639
	n=100000	112.47	1.615
	n=210000	249.529	18.783
	n=256000	297.419	10.063
	n=500000	561.674	8.785
	n=1000000	1161.6	37.02
Argon2id	*t=1, m=16 MiB*	15.349	1.458
	t=1, m=19 MiB	17.971	0.439
	t=2, m=19 MiB	26.054	1.485
	t=3, m=12 MiB	22.087	1.849
	t=1, m=46 MiB	39.494	2.654
	t=2, m=46 MiB	65.589	3.86
	t=1, m=64 MiB	55.599	2.365
	t=2, m=64 MiB	93.253	4.461
	t=3, m=64 MiB	132.288	7.504
	t=2, m=128 MiB	205.185	5.202

B. Multi-Process Dictionary Attack Simulation Results

The second experiment evaluated hardware scaling limitations by simulating a dictionary attack which utilizes both a single-threaded and multi-process approach. Table 5. details the measured throughput in the form of password evaluated per second and the resulting parallel speedup ratio.

TABLE 5. SIMULATED DICTIONARY ATTACK RESULTS

KDF	Configuration	Total Time (s)	Throughput (passwords/s)	ETA (hours)
PBKDF2	*n=64000, single-threaded*	74.34	13.112	303.9
	n=64000, multi-process	26.29	38.074	104.7
	n=210000, single-threaded	258.84	3.867	1030.3
	n=210000, multi-process	92.71	10.797	369.1
	n=256000, single-threaded	314.68	3.181	1252.6
	n=256000, multi-process	121.27	8.254	482.7
	n=1000000, single-threaded	1198.25	0.835	4769.7
	n=1000000, multi-process	437.23	2.289	1740.4
Argon2id	*t=1, m=46 MiB, single-threaded*	39.61	25.272	157.7
	t=1, m=46 MiB, multi-process	16.9	59.226	67.3
	t=2, m=19MiB, single-threaded	27.6	36.264	109.9
	t=2, m=19 MiB, multi-process	11.65	85.936	46.4
	t=2, m=64 MiB, single-threaded	96.52	10.371	384.2
	t=2, m=64 MiB, multi-process	40.05	24.992	159.4

C. Database Initialization Overhead Results

The final experiment evaluated the raw performance of both KDFs and translated it into practical software engineering metrics through the operational context of SQLCipher. The pipeline encompasses both the KDF execution and the following AES-256-CBC decryption of the initial database header page. Table 4. details the user-perceived latency of the initial startup phase, while Table 6. outlines the baseline query throughput measured in MiB per second.

TABLE 6. DATABASE COLD START RESULTS

KDF	Configuration	KDF Mean Time (ms)	Decrypt Time (ms)	Total Cold Start (ms)
PBKDF2	*n=64000*	79.695	0.094	79.789
	n=256000	323.251	0.258	323.51
	n=500000	634.958	0.119	635.076
	n=1000000	1202.799	0.098	1202.897
Argon2id	*t=1, m=46 MiB*	40.242	0.083	40.325
	t=2, m=46 MiB	68.304	0.079	68.382
	t=2, m=64 MiB	94.843	0.096	94.94
	t=3, m=64 MiB	140.144	0.082	140.226

TABLE 7. DATABASE QUERY THROUGHPUT RESULTS

Pages	Total Bytes	Elapsed Time (ms)	Throughput (MiB/s)
500	*2048000*	5.328	366.6

V. DISCUSSION

The results gathered from the three experiments present a comprehensive view of the performance properties and security tradeoffs between PBKDF2 and Argon2id as key derivation functions within the operational context of SQLCipher. This section interprets those results within the security and usability context established.

A. KDF Benchmark Analysis

The results shown in Table 4. confirms that PBKDF2 has a linear relationship between its iteration count and derivation time. Each increment in iteration count corresponds to a near-proportional increase in latency. For example, increasing the iteration count from 64,000 to 256,000 produced an increase in latency from 71.772ms to 297.419ms, reflecting a rough 4.1x proportional increase. Although this linearity is predictable, it provides a fundamental limitation in security, where greater security in PBKDF2 is acquired only through greater user-perceived latency with no alternative parameter to adjust.

On the other hand, Argon2id exhibits multi-dimensional scaling behavior as demonstrated by the combination of both time and memory cost. For example, holding time cost at *t=1* while increasing memory from 46 MiB to 64 MiB resulted in an increase of latency from 39.494ms to 55.599ms, a ~40.8% increase. Another example is increasing the time cost from *t=1* to *t=3* while holding the memory cost at 64 MiB resulting in a 137.9% increase in latency. This implies that the time cost parameter alone carries a stronger influence on raw CPU execution time than its memory cost, as the memory access within a single pass benefit from the CPU's cache and does not add proportional overhead. Interestingly, Argon2id's minimum configuration at *t=1* and *m=46 MiB* derived a key roughly 7.53

times faster than SQLCipher's default configuration for PBKDF2 at 256,000 iterations. While this may suggest Argon2id as the less secure option, the opposite can be argued to be true when accounting for memory-hardness and its implications for parallelized attacks, both of which are addressed by the following section.

B. *Implication of Multi-Process Attacks*

Table 5. shows the parallel scaling results between the two algorithms. Across all the tested configurations, PBKDF2 consistently had a higher parallel speedup ratio than Argon2id when scaling from single-threaded to an 8-worker execution. PBKDF2's higher scaling efficiency is a direct consequence of its CPU-bound, memory-light design, where each worker process requires minimal memory per instance. This allows more parallel processes to operate concurrently without the hindrance of memory contention.

PBKDF2's scaling behavior carries a critical implication when evaluated in the context of real-world attacks. The raw CPU-bound estimated time of completion comparison certainly appears to favor PBKDF2 at certain configurations, such as the default SQLCipher configuration at *n=256,000* iterations resulting in an estimate of 482.7 hours as opposed to Argon2id at *t=2*, *m=64 MiB* at 159.4 hours. However, this comparison is misleading within the threat model discussed in this paper. The 8-worker simulation reflects CPU-level parallelism that does not represent the magnitude of parallelism available to a GPU attacker. This comes back to the consequences of PBKDF2's design as a CPU-bound and memory-light algorithm. Because of it, its throughput scales nearly linearly with GPU core count, which could number in the thousands. In contrast, Argon2id enforces a per-instance memory requirement that constrains the number of concurrent instances a GPU can sustain, regardless of the number of cores it has. The lower parallel scaling efficiency observed in the second experiment reflects Argon2id's property that limits GPU acceleration.

Among Argon2id configurations, the variant *t=2*, *m=64 MiB* gave the best outcome, achieving the lowest throughput at 24.992 passwords/s and the longest ETA at 159.4 hours. Comparing it with the other two configurations reveals that the memory cost gives a stronger effect on limiting throughput as opposed to the time cost. This aligns with the theoretical framework, where an increase in memory usage would deter the effects of parallel computation.

C. *Overhead Analysis*

The cold start results in Table 6. addresses the practical paradox established in the introduction regarding Argon2id's security tradeoff with unacceptable application latency. In short, the results demonstrate that Argon2id can provide security benefits whilst yielding favorable user-perceived latency. Each Argon2id configuration remains well below the 150ms threshold for acceptable user-perceived latency, with the strongest tested configuration of *t=3*, *m=64 MiB* resulting in only 140.226ms of total cold start time. This gave a strong contrast with PBKDF2, where the SQLCipher's default configuration of *n=256,000* iterations resulted in 323.51ms, while its strongest configuration of *n=1,000,000* exceeded one second at 1,202.897ms. PBKDF2's results render it less suitable than Argon2id for applications requiring very high performance.

Across all configurations, the decryption process of the database header page involving AES-256-CBC practically contributed negligibly towards the total cold start time, with all record values remaining below 0.3ms. This confirms that KDF latency basically makes up the entirety of the latency cost in database usage. Furthermore, the query throughput measurement of 366.6 MiB/s which is independent of KDF configuration confirms that key derivation is strictly a one-time penalty and has no effect on following query performance, since the derived encryption key is stored in the memory as opposed to deriving it every time a query is performed.

VI. CONCLUSION

All three experiments provided a comprehensive comparison between PBKDF2 and Argon2id's feasibility as a KDF for SQLCipher, evaluating both KDFs' derivation latency, resistance to parallelized dictionary attacks, and practical application startup overhead. The benchmarking experiment established that PBKDF2's derivation time scales linearly with its iteration count, leaving it as the sole parameter which constitutes the algorithm's level of security. On the other hand, Argon2id offers a multi-dimensional parameter space which allows greater potential in formulating a stronger combination. The dictionary attack simulation showed that PBKDF2 exhibits higher parallel scaling efficiency than Argon2id across multiple processing cores. This property would be further amplified under GPU-accelerated attacks, where PBKDF2's memory-light design allows a near-linear throughput scaling, compared to Argon2id's memory-hard design which minimizes parallel computation scaling. The final experiment regarding database overhead experiment confirmed that all evaluated Argon2id configurations remain below the acceptable 150ms latency, with PBKDF2 configurations struggling to keep up. Collectively, these results refute the concern over Argon2id's memory and processing requirements as unsuitable for SQLCipher deployments. Argon2id provides a stronger resistance to parallelized attacks whilst also keeping user-perceived latency at the minimum than the PBKDF2 at roughly equivalent configurations. Both traits allow Argon2id to be a suitable candidate for replacing PBKDF2 as SQLCipher's default key derivation function.

It should be noted that the dictionary attack simulation conducted was executed exclusively in a CPU-bound environment due to device limitations, which leaves Argon2id's GPU resistance to be argued through analysis rather than measured empirically. Further research could extend this analysis through GPU-accelerated benchmarks using tools such as the Hashcat password cracking tool to validate the scaling gap between the two algorithms under actual attack conditions. Moreover, testing on mobile devices which offers limited resources could further clarify Argon2id's practical feasibility in mobile SQLCipher deployments.

ACKNOWLEDGMENT

The author expresses profound gratitude to God for thecompletion of this paper. The author also expresses

gratitude to Prof. Dr. Ir. Rinaldi, M.T, the lecturer of II4021 Cryptography course, for teaching and providing the author's knowledge in the making of this paper, as well as guiding the author to completing the course this semester.

REFERENCES

[1] OWASP Mobile Security Team. "OWASP Mobile Top 10." The OWASP Foundation. https://owasp.org/www-project-mobile-top-10/ (accessed Jun. 17, 2026).

[2] P. Tippe, M. P. Berner, "Evaluating Argon2 Adoption and Effectiveness in Real-World Software," 2025, doi: https://doi.org/10.48550/arXiv.2504.17121.

[3] R. Munir. "Fungsi Hash." Website Rinaldi Munir. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2025-2026/26-Fungsi-hash-2026.pdf (accessed Jun. 18, 2026).

[4] R. Munir. "Beberapa Fungsi Hash." Website Rinaldi Munir. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2025-2026/26-Fungsi-hash-2026.pdf (accessed Jun. 19, 2026).

[5] H. Rhim. "What Are Key Derivation Functions?." Baeldung. https://www.baeldung.com/cs/kdf-cryptography (accessed Jun. 18, 2026).

[6] A. Visconti, S. Bossi, H. Ragab, and A. Calò, "On the weakness of PBKDF2," *IACR Cryptol. ePrint Arch.*, vol. 2016, pp. 273, 2015, Available: https://visconti.di.unimi.it/PBKDF2.pdf.

[7] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," IETF, RFC 2898, Sep. 2000, Available: https://datatracker.ietf.org/doc/html/rfc2898#section-5.2.

[8] A. Visconti, O. Mosnáček, M. Brož, and V. Matyáš, "Examining PBKDF2 security margin—Case study of LUKS," *Journal of Information Security and Applications*, vol. 46, pp. 296-306, 2019, doi: https://doi.org/10.1016/j.jisa.2019.03.016.

[9] M. Preziuso. "Password Hashing: Scrypt, Bcrypt and ARGON2." Medium. https://medium.com/@mpreziuso/password-hashing-pbkdf2-scrypt-bcrypt-and-argon2-e25aaf41598e (accessed Jun. 18, 2026).

[10] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications," RFC 9106, Sep. 2021, Available: https://www.rfc-editor.org/rfc/rfc9106.html.

[11] S. Lombardo and N. Parker. "Enhancing Password Based Key Derivation Techniques." Zetetic. https://www.zetetic.net/storage/files/enhancing-password-based-key-derivation-techniques.pdf (accessed Jun.19, 2026).

[12] U. Telle. "SQLCipher: AES 256 Bit." SQLite3 Multiple Ciphers. https://utelle.github.io/SQLite3MultipleCiphers/docs/ciphers/cipher_sqlcipher/ (accessed Jun. 19, 2026).

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Aria Judhistira, 13523112