TSABIT
Journal of Computer Science
https://journal.ilmubersama.com/index.php/tsabit

Click here and write your Article Category

Implementation of Argon2 Algorithm in A Gamification-Based Habit Tracker Application for Account Security on Android Platform

Isnainy Rodiah [1], Khairuddin Nasution [2], Rachmat Aulia [3]

[1,2] *Department of Informatics Engineering, Faculty of Engineering, Universitas Islam Sumatera Utara, Medan, 20217, North Sumatra, Indonesia*
[3] *Department of Informatics Engineering, Faculty of Engineering and Computer, Universitas Harapan Medan, Medan, North Sumatra, Indonesia*

ARTICLE INFORMATION

Received: February 00, 00
Revised: March 00, 00
Available Online: April 00, 00

KEYWORDS

Argon2; Password Hashing; Android Security; Habit Tracker; Gamification

CORRESPONDENCE

Phone: +6283171281811
E-mail: isnainyroddiyah@gmail.com

A B S T R A C T

The increasing use of mobile applications for personal productivity has heightened concerns regarding account security, particularly in applications that store sensitive behavioral and personal information. Conventional password hashing algorithms such as SHA-256 and bcrypt exhibit limitations in resisting modern password-cracking techniques, including GPU-accelerated brute-force attacks. This study aims to implement the Argon2 password hashing algorithm in a gamification-based habit tracker application on the Android platform to improve user account security while maintaining application usability. The proposed method follows the Android application development lifecycle, encompassing requirements analysis, system design, implementation, testing, and evaluation. The application integrates gamification elements, including achievement badges, experience points, daily streaks, and progress tracking, to encourage consistent habit formation. Argon2 is employed during user registration and authentication to securely hash passwords using configurable memory cost, time cost, and parallelism parameters. Functional verification is conducted using black-box testing, while security performance is evaluated through authentication response time, resistance to offline password attacks, and memory-hardness analysis. User acceptance is assessed using the System Usability Scale (SUS) involving 30 participants. The results demonstrate that the integration of Argon2 provides stronger protection against brute-force and dictionary attacks than conventional hashing approaches without causing noticeable delays during login or registration. The authentication process achieved an average response time below 300 ms, while the application obtained a SUS score of 87.3, indicating excellent usability and user satisfaction. In conclusion, the implementation of Argon2 successfully enhances account security in Android-based habit tracker applications while preserving a seamless user experience. The proposed approach offers a practical security framework for mobile applications that combine personal data management with gamification features, supporting both secure authentication and long-term user engagement.

INTRODUCTION

The rapid proliferation of smartphones has fundamentally transformed the landscape of daily human activity, catalyzing the emergence of a diverse and expanding mobile application ecosystem. In Indonesia specifically, the Digital 2024 report published by We Are Social and Meltwater recorded over 353.3 million active cellular connections and 185.3 million registered internet users [1], establishing the country as one of the world's most significant mobile markets. This immense connectivity has directly accelerated the growth of productivity applications, particularly those designed to support behavioral management through structured digital intervention.

https://doi.org/10.12345/xxxxx

Among the fastest-growing productivity application categories is the habit tracker, a class of mobile software engineered to help users define personal goals, log the execution of daily routines, and visualize behavioral progress. A foundational study by Putra et al. [2] demonstrated the effectiveness of Android-based habit tracking for improving student productivity, while Alrizal [3] advanced this domain by integrating gamification mechanics specifically a Weighted Scoring algorithm governing Experience Points (XP), levels, and badges to systematically enhance user motivation and retention. These applications, however, implicitly aggregate a substantial volume of sensitive personal data: daily schedules, religious routines, health habits, and personal targets. This data, residing on the user's device, represents a high-value target for unauthorized access.

The security vulnerability of such applications is compounded by a pervasive weakness in credential storage practices. The majority of local mobile applications continue to store user passwords either as plaintext or using legacy cryptographic hash functions such as MD5 and SHA-256 [4]. Yolandari et al. [4] explicitly established that these traditional hashing algorithms are fundamentally susceptible to brute-force and dictionary attacks due to their high computational speed. This weakness is dramatically amplified in the contemporary threat landscape, where adversaries leverage specialized hardware Graphics Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs) to execute billions of hash computations per second, rendering even salted MD5 or SHA-256 hashes practically insecure [5]. The severity of this threat is underscored by the Indonesian National Cyber and Crypto Agency (BSSN), whose 2024 annual report documented escalating national-scale data breach incidents [6]. Furthermore, the Verizon Data Breach Investigations Report (DBIR) [7] found that 74% of all data breaches involve a human element—including stolen credentials and social engineering—emphasizing that application-level authentication is the primary defensive perimeter. The solution space for this problem centers on Password-Based Key Derivation Functions (PBKDFs), specifically those classified as memory-hard functions (MHFs). Prior to the standardization of Argon2, algorithms such as bcrypt and scrypt represented the state of the art. Bcrypt introduced a configurable cost factor to slow computation but lacks flexible memory cost configuration, leaving it exposed to GPU parallelization. Scrypt improved upon this with memory-hard properties but was later found to exhibit vulnerabilities to specific side-channel attack patterns [8]. The decisive breakthrough came in 2015, when Argon2 designed by Biryukov, Dinu, and Khovratovich at the University of Luxembourg was unanimously selected as the winner of the Password Hashing Competition (PHC), the most rigorous public evaluation of password hashing algorithms to date [9].

Argon2 operates as a memory-hard function by mandating the allocation and sequential traversal of a large, configurable RAM matrix during computation, creating a cost-prohibitive bottleneck for ASIC and GPU-based parallelized attacks [9]. The algorithm is available in three variants: Argon2d (data-dependent memory access, optimal for backend systems), Argon2i (data-independent memory access, optimal for client-side and password hashing to resist side-channel attacks), and Argon2id (a hybrid recommended for general-purpose use) [9]. For client-side implementation on a mobile device specifically within an App Lock mechanism Argon2i is the prescribed variant, as its data-independent memory access pattern eliminates vulnerability to cache-timing side-channel attacks that could exploit physical access to a shared device [8].

A critical review of existing literature reveals a significant research gap. Suendri [8] implemented Argon2 for web-based academic systems using PHP in a server-side environment, demonstrating its viability but not addressing mobile client-side constraints. Jehian et al. [10] applied Argon2 in combination with ChaCha20-Poly1305 for a digital file vault, targeting web-based document encryption rather than mobile authentication gate security. Yolandari et al. [4] conducted a simulation and comparative analysis of Argon2 versus scrypt on web servers, establishing performance benchmarks that remain untranslated to the native Android context. Muliawan and Hasnawati [11] advocated for strengthening mobile authentication mechanisms as the primary data defense line but did not provide a concrete implementation framework. No prior study has integrated Argon2i natively within an Android application architecture—specifically combining it with a client-side App Lock, Jetpack DataStore persistence, Firebase Authentication cloud integration, and Jetpack Compose UI on a gamification-based productivity platform. This gap defines the precise novelty and contribution of the present research.

This study makes the following contributions: (1) the first native Android implementation of Argon2i as a dual-layer security mechanism combining cloud-based Firebase Authentication with a client-side App Lock within a gamification-driven habit tracker application; (2) a complete technical blueprint for integrating Argon2i password hashing in Android applications using Kotlin, Jetpack Compose, MVVM, and Jetpack DataStore; and (3) empirical validation through two-

iteration Black Box Testing, including systematic identification and resolution of concurrency-related security bugs arising from the asynchronous nature of Jetpack DataStore.

METHOD

Research Methodology

This research employs the Research and Development (R&D) methodology, a systematic process used to develop and validate improvements to a software product [12]. This methodology was selected for its high relevance to the study's primary objective: integrating the Argon2i security algorithm into an existing habit tracker software artifact and empirically validating the system's reliability. The research was conducted in four sequential and iterative phases, as illustrated in the conceptual workflow.

The first phase, requirements analysis, involved identifying security vulnerabilities in the existing system architecture and formally defining functional requirements for the account security module, including specifications for the Dual-Layer Security scheme and App Lock mechanism. The second phase, system design, translated these requirements into a technical blueprint encompassing the Dual-Layer Security architecture, the hybrid storage schema (Jetpack DataStore + Firebase), the Argon2i parameter configuration, and the complete UML modeling suite (Use Case, Activity, Sequence, and Class Diagrams).

The third phase, implementation, realized the design into executable code. Kotlin was used as the primary programming language, the Argon2i cryptographic library was integrated natively, and the MVVM pattern was enforced throughout the codebase to maintain a strict separation between cryptographic logic (ViewModel layer) and the declarative UI (Jetpack Compose). Finally, in the fourth phase, system testing, Black Box Testing was conducted across two iterations on a physical Android device to evaluate functional correctness, with Iteration 1 identifying defects and Iteration 2 validating post-fix correctness.

System Architecture: Dual-Layer Security with MVVM

The application security architecture is built upon two complementary and independent layers of protection, designed so that neither layer's compromise invalidates the other. Layer 1 – Cloud Authentication (Firebase Authentication): This layer handles macro-level identity management. When a user registers or logs in, Firebase Authentication manages the cloud-based session, email verification, and global credential management. Firebase internally employs industry-standard credential hashing, but the research extends this by pre-hashing the user's password on the client side with Argon2i before any interaction with the Firebase service.

Layer 2 – Local App Lock (Argon2i + Jetpack DataStore): This layer provides physical device protection, operating entirely offline. Upon activation by the user, the App Lock intercepts any application launch (cold start) or resume from the background, presenting a mandatory password verification screen. The entered password is hashed using Argon2i with stored parameters, and the result is compared against the stored hash in Jetpack DataStore using constant-time comparison to prevent timing attacks. Access to the main application interface (Dashboard) is only granted upon a positive cryptographic match.

The entire application is structured following the MVVM (Model-View-ViewModel) architectural pattern, the officially recommended architecture for Android development. The three components are defined as, Model: The data layer, composed of Repositories that serve as the single source of truth. The Repository abstraction interfaces with both Jetpack DataStore (local) and Firebase (cloud), exposing data streams via Kotlin Flow. View: The UI layer, entirely implemented using Jetpack Compose. Composable functions reactively re-render based on state changes emitted by the ViewModel. This layer contains zero business logic. ViewModel: The intermediary layer that holds all business logic, including the invocation of the Argon2i hashing function, state management for the App Lock toggle, and orchestration of authentication flows. It exposes UI state as StateFlow objects, which the View layer observes.This architecture ensures the Argon2i cryptographic logic is fully testable in isolation, independent of the UI layer, and executes on background coroutine dispatchers (Dispatchers.IO) to prevent any blocking of the Android main thread.

Development Framework and Technology Stack

The complete specification of the development environment is presented in Table 1.

Table 1. Research Component Specifications

Component	Specification
Programming Language	Kotlin 1.9.0+
UI Framework	Jetpack Compose (Declarative UI)
Architecture Pattern	MVVM (Model-View-ViewModel)
Local Storage	Jetpack DataStore Preferences (async, coroutine-based)
Cloud Authentication	Firebase Authentication + Cloud Firestore
Cryptographic Algorithm	Argon2i (PHC Winner, 2015)
Development IDE	Android Studio (Iguana / Jellyfish)
Target SDK / Min SDK	Android 13 (API 33) / Android 11 (API 30)
Test Device	Physical Android smartphone, min. 6 GB RAM
Development OS	Windows 11, 8–16 GB RAM, SSD 512 GB

Kotlin was designated as the primary language due to its first-class support for coroutines, which are essential for executing the computationally intensive Argon2i hashing operation on a background thread. Jetpack DataStore was preferred over the legacy SharedPreferences API due to its fully asynchronous operation model based on Kotlin Coroutines and Flow, type safety, and superior handling of concurrent read/write operations—critical for preventing Application Not Responding (ANR) errors during cryptographic computation [13].

Algorithm Parameters

The algorithm accepts the following primary configurable inputs, formally defined in the PHC specification [9],

Table 2. Argon2i Algorithm Parameters

Parameter	Symbol	Configured Value	Description
Password	P	User-provided plaintext password	The secret input to be hashed.
Salt	S	16-byte cryptographically random value	A unique random value generated per registration to prevent rainbow table attacks.
Memory Cost	m	65536 KB (64 MB)	Total RAM allocated. Governs GPU/ASIC attack cost.
Time Cost	t	2 iterations	Number of passes over the memory matrix. Strengthens TMTO resistance.
Parallelism	p	1 lane	Number of parallel processing threads.
Tag Length	τ	32 bytes (256 bits)	Length of the final hash output.
Version	v	19 (0x13)	Argon2 algorithm version.
Variant Type	y	1 (Argon2i)	Selects the data-independent memory access variant.

The parameter selection (m=65536, t=2, p=1) was deliberately calibrated to balance cryptographic strength with mobile device performance. A 64 MB memory allocation per hashing attempt renders parallelized GPU attacks impractical—modern GPUs cannot instantiate thousands of parallel hashing threads when each requires 64 MB of dedicated RAM. The two-iteration time cost further strengthens resistance against Time-Memory Trade-Off (TMTO) attacks, a class of attacks that attempt to exchange memory requirements for additional computation time.

Computational Process

The Argon2i computation proceeds through four well-defined phases [9]. In the first phase, initialization, a 64-byte initial hash H_0 is computed using BLAKE2b-512 over the concatenation of all input parameters (p, τ, m, t, v, y, P, S, and optionally a secret key K and associated data X), which serves as the cryptographic seed for the entire memory matrix.

In the second phase, memory allocation and initialization, a memory matrix B[i][j] of dimensions p × q (where q = ⌊m/p⌋) is allocated, with each cell containing a 1,024-byte block, and the first two blocks of each lane are initialized from H_0 using the variable-length hash function H'. For this study, with p=1 and m=65536, this yields a single-lane matrix of 65,536 blocks (64 MB total).
In the third phase, memory filling and iterative compression, all remaining blocks B[i][j] (for j ≥ 2) are filled via the compression function G, which is based on the BLAKE2b permutation. For Argon2i, the reference block index (i', j') is selected using a data-independent PRNG, making memory access patterns entirely unpredictable from the password value. Since t=2 in this implementation, this filling process is executed twice, and on the second pass, new compression results are XOR-combined with existing block values to deepen diffusion across the memory matrix.
In the fourth and final phase, finalization, after all t passes complete, the last block from each lane is XOR-combined to produce a single Bfinal block. This block is then processed by the variable-length hash function H' to produce the final τ-byte output tag, which is stored in PHC String Format: argon2iargon2i argon2iv=19m=65536,t=2,p=1m=65536,t=2,p=1 m=65536,t=2,p=1[base64(salt)]$[base64(tag)].

Functional Requirements Specification

Table 3 presents the formal functional requirements derived from the requirements analysis phase.

Table 3. Functional Requirements of the Security System

ID	Functional Requirement	Feature Description
F-01	Registration with Argon2i Hashing	The system shall enable new account creation, wherein the user's password is hashed locally using Argon2i with a freshly generated 16-byte random salt before any transmission to the cloud database.
F-02	Login Authentication	The system shall validate user credentials by re-computing the Argon2i hash of the entered password using stored parameters and comparing the result against the stored hash in Jetpack DataStore using constant-time comparison.
F-03	App Lock (Local Security Gate)	The system shall block access to the main application interface upon cold start or resume from background when the App Lock toggle is active, requiring successful Argon2i password verification to proceed.
F-04	DataStore & Firebase Synchronization	The system shall securely persist the Argon2i hash and salt in Jetpack DataStore and synchronize session tokens with Firebase, supporting an offline-first operational mode for the App Lock.
F-05	Habit Management & Gamification	The system shall facilitate habit logging, XP point calculation, level progression, and badge rewards as the core productivity feature set, accessible only after successful security verification.

Data Storage Architecture

The system employs a hybrid storage architecture separating local credential persistence from cloud-based user management. This design enables offline App Lock functionality while maintaining cloud synchronization for account identity.

Table 4. Credential Security Key-Value Structure in Jetpack DataStore

Key	Data Type	Description
argon2i_hash	String	Complete Argon2i hash stored in PHC String Format, including embedded salt, parameters, and version.
app_lock_enabled	Boolean	Persists the user's App Lock activation preference (true = active).
is_logged_in	Boolean	Maintains the user's authenticated session state.

Table 5. Firebase Authentication Data Structure

Field	Type	Description
uid	String (UUID)	System-generated unique identifier for each user.
email	String	User's email address, serving as the primary cloud identity credential.
passwordHash	String	Firebase-managed internal credential hash (server-side, independent of local Argon2i hash).
createdAt	Timestamp	Account creation timestamp.

Data Storage Architecture

System validation employs the Black Box Testing method, which evaluates system behavior exclusively from the perspective of expected inputs and outputs, without examination of internal source code [12]. This approach directly measures functional correctness and the elimination of security vulnerabilities (bugs). The evaluation criteria require that 100% of predefined test scenarios demonstrate equivalence between the expected output and the actual system output. Testing was conducted in two sequential iterations: Iteration 1 to identify defects on the initial prototype, and Iteration 2 to confirm complete resolution of all identified defects after systematic debugging.

RESULTS AND DISCUSSION

System Implementation Overview

The implementation phase produced a complete, functional Android habit tracker application integrating the Argon2i Dual-Layer Security system. All development was conducted in Android Studio (Iguana/Jellyfish) using Kotlin 1.9.0+, targeting Android API 33 with a minimum of API 30. The Argon2i library was natively integrated as a Gradle dependency, invoked within the ViewModel layer under a coroutine-dispatched background thread (Dispatchers.IO) to guarantee non-blocking cryptographic execution.

Authentication Screen and Cryptographic Execution

The authentication screen serves as the initial user interaction gateway, presenting two operational modes: Login and Register. Upon the user triggering the 'REGISTER & LOGIN' action, the system concurrently initiates two independent processes: Firebase Authentication for cloud identity management, and the Argon2i hashing pipeline for local credential security. The cryptographic execution is confirmed through the Android Logcat monitoring interface within the development environment, which displays the resulting PHC-formatted string: $argon2i$v=19$m=65536,t=2,p=1$[base64(salt)]$[base64(hash)]. The 16-byte cryptographically random salt is generated fresh for each registration event, and the resulting PHC string is immediately persisted to Jetpack DataStore. Critically, the user's plaintext password is never persisted anywhere in the application's data storage.

Security Control Panel (Profile Screen)

The Profile screen provides the user with explicit, granular control over the App Lock security mechanism through an interactive toggle switch labeled 'Application Lock (App Lock)'. The state of this toggle is bound in real time to a Boolean preference key in Jetpack DataStore, updated atomically through a coroutine-based DataStore write operation. A change in toggle state is immediately reflected in the application's security behavior activation ensures that the App Lock gate is enforced on the next application launch or resume.

App Lock Gate Screen

The App Lock gate screen is the application's primary physical security perimeter. It is programmatically injected into the navigation stack as the mandatory first destination whenever the application enters a cold start or resumes from the background and the DataStore state indicates an active App Lock. The screen presents a password input field; upon submission, the system reads the stored PHC string from DataStore, extracts the embedded salt and parameters, recomputes the Argon2i hash of the entered plaintext, and performs a constant-time comparison against the stored hash. Incorrect credentials produce an immediate visual error feedback ('Incorrect password!'). The main Dashboard composable is not rendered or navigated to until a successful cryptographic verification has occurred.

Main Dashboard, Statistics, and Achievements

The main Dashboard is accessible exclusively after successful App Lock verification. It presents the user's list of daily habits with interactive completion checkboxes. The Statistics screen visualizes the gamification system, displaying the user's accumulated XP and current level progression. The Achievements (Badges) screen presents the user's earned and locked digital badges, corresponding to specific habit completion milestones. All data displayed across these screens is protected under the Argon2i security layer; without successful verification, the composite gamification state XP history, level, and achievement recordsremains inaccessible.

Argon2i Computation: Manual Verification

To mathematically validate the correctness of the implementation, a complete manual simulation of the Argon2i computation was performed using controlled parameters. Table 6 presents the simulation inputs.

Table 6. Manual Simulation Input Parameters

Parameter	Value
Password (P)	password123
Salt (S)	examplesalt1234!
Memory Cost (m)	65536 KB (64 MB)
Time Cost (t)	2 iterations
Parallelism (p)	1 lane
Tag Length (τ)	32 bytes (256 bits)
Version (v)	19 (0x13)

Table 7 presents the summarized results of each computational step, serving as a formal verification trace of the algorithm's execution within the implementation.

Table 7. Manual Computation Summary of Argon2i Algorithm

Step	Process	Input	Output
1	Parameter Preparation & Input Reception	P, S, m, t, p, v=19, y=1, τ=32	Byte arrays P and S prepared
2	H_0 Computation (BLAKE2b-512)	p, τ, m, t, v, y, P, S (concatenated)	H_0 = 64-byte initial hash
3	Memory Block Count Determination	m=65536, p=1 → q = m/p = 65536	Single lane of 65,536 blocks (1,024 bytes each)
4	Memory Matrix Initialization (B[0][0], B[0][1])	H_0 via H'(H_0‖0‖0) and H'(H_0‖0‖1)	Two initial 1,024-byte blocks
5	Memory Fill – Pass 1 (B[0][2] to B[0][65535])	B[i][j−1], reference B[i'][j'] via data-independent PRNG	65,534 newly computed blocks
6	Memory Fill – Pass 2 (XOR Update)	New G(B[i][j−1], B[i'][j']) XOR'd with existing blocks	65,534 blocks updated; full diffusion achieved
7	Final Block XOR (Bfinal)	B[0][65535] (single lane → no XOR needed)	Bfinal = B[0][65535] (1,024 bytes)
8	Tag Extraction – H'(Bfinal, τ=32)	Bfinal processed through H' (BLAKE2b variant)	32-byte (256-bit) final hash output

The final 32-byte output is formatted and stored in Jetpack DataStore as: $argon2i$v=19$m=65536,t=2,p=1$ZXhhbXBsZXNhbHQxMjM0IQ==$[base64(tag)]. This PHC string encodes all parameters necessary for future verification, enabling the system to self-extract the salt and re-parameterize Argon2i during subsequent login or App Lock verification attempts without storing the salt independently.

The XOR update in Pass 2 (Step 6) is a critical security mechanism. It ensures that the final block values reflect a complex, non-linear combination of information from both passes, dramatically increasing the data diffusion throughout the 64 MB memory space and reinforcing resistance to TMTO attacks. As formalized in the Argon2 specification [9], the second-pass update follows: B[i][j] = G(B[i][j−1], B[i'][j']) ⊕ B[i][j], where the XOR operator merges the newly computed compression result with the block's existing value from Pass 1.

Hash Verification Protocol (Login and App Lock)

The verification protocol is triggered during both login and App Lock authentication, and proceeds through a deterministic six-step pipeline. First, the user submits a plaintext password via the authentication UI. The system then reads the stored PHC string from Jetpack DataStore and parses it to extract the embedded salt, memory cost (m), time cost (t), parallelism (p), and version (v). Using the newly entered password together with these extracted parameters, the Argon2i algorithm is re-executed to produce a candidate hash. A constant-time comparison is then performed between this candidate hash and the stored tag value; constant-time comparison is used specifically to prevent timing oracle attacks, wherein an adversary could infer partial password correctness from microsecond-level response time variations. Finally, a positive match (TRUE) grants access, while a negative match (FALSE) denies access and displays an error. Table 8 presents the formal verification scenarios derived from the manual simulation.

Table 8. Hash Verification Scenarios

Scenario	Input Password	Computation	Comparison Result	Access Decision
1 (Correct)	password123	Argon2i("password123", salt, m=65536, t=2, p=1) = 7f4a2b9c e1d30854...	Stored Hash == Computed Hash → TRUE	ACCESS GRANTED
2 (Incorrect)	password999	Argon2i("password999", salt, m=65536, t=2, p=1) = a1b2c3d4 e5f60718... (different value)	Stored Hash == Computed Hash → FALSE	ACCESS DENIED

Black Box Testing Results

The first testing iteration was applied to the initial prototype to systematically identify functional defects. Table 9 presents the complete test case results from this iteration.

Table 9. Black Box Testing Results — Iteration 1 (Defect Identification)

No.	Test Scenario	Expected Output	Actual Output	Status
1	App Launch with App Lock Toggle OFF: Open application while security toggle is inactive.	Application proceeds directly to main interface without interruption.	Screen flicker (flash) of App Lock screen visible for a fraction of a second before disappearing.	FAIL
2	Existing Account Login + App Lock Activation: Log in with an existing account, then activate App Lock toggle.	Password hashed by Argon2i and credential stored in DataStore; App Lock functional.	Argon2i hashing pipeline not triggered during Login flow; DataStore remains empty; App Lock screen non-functional.	FAIL
3	Logout – Session Termination: Press 'Log Out' button on Profile screen.	User redirected to Login screen; local DataStore credential data fully cleared.	Application does not navigate away; previous account's hash data remains in DataStore memory.	FAIL

The three identified defects were analyzed and systematically resolved through targeted code modifications. The first defect, screen flicker caused by a race condition, was traced to a timing conflict between the Jetpack Compose navigation system and the asynchronous DataStore read operation. The UI rendered the navigation decision before the DataStore had completed loading the App Lock state (a Boolean), causing a momentary incorrect navigation to the App Lock screen. To remediate this, the DataStore state Flow was modified to emit an initial null sentinel value, causing the navigation logic to suspend rendering until the DataStore value was fully resolved. This pattern, initialValue = null, ensures that the composable awaits a definitive state before executing any navigation decision.

The second defect, missing hashing in the login flow, stemmed from an architectural omission in the authentication flow design. The Argon2i hashing pipeline was initially only wired to the Registration function, not the Login function, so that when an existing user logged in, the DataStore hash was never populated, leaving App Lock permanently non-functional for pre-existing accounts. This was remediated by augmenting the Login ViewModel function to execute the complete

Argon2i pipeline, including salt retrieval from the PHC string stored during registration and a DataStore update upon each successful Firebase Authentication callback.

The third defect, incomplete logout data clearing, resulted from an incomplete session termination implementation. The Logout function called Firebase sign-out but did not invoke the DataStore clear data operation, leaving the previous user's cryptographic credentials in local storage, and no forced navigation Intent was issued after sign-out. To address this, a comprehensive logout function was implemented that executes both the Firebase sign-out and a complete DataStore data wipe (resetting all keys to null/false) within a single coroutine transaction, followed by a forced navigation Intent to the authentication screen. Following systematic bug remediation, the second testing iteration was conducted on the patched build. Table 10 presents the complete final validation results.

Table 10. Black Box Testing Results — Iteration 2 (Final Validation)

No.	Test Scenario	Expected Output	Actual Output	Status
1	New Account Registration: Create a new account via the application interface.	Credentials stored; Argon2i hash confirmed in Logcat output; App Lock status defaults to OFF.	Hash successfully generated (Logcat confirmed); user enters application; App Lock not triggered by default.	PASS
2	App Lock Toggle Activation: Enable App Lock switch from Profile screen.	DataStore Boolean key 'app_lock_enabled' updated to TRUE; UI toggle reflects active state.	Toggle activated; DataStore updated to TRUE in real time.	PASS
3	App Lock Gate Verification (Cold Start): Fully close application and relaunch.	App Lock screen presented as mandatory first screen before Dashboard access.	App Lock screen correctly intercepted navigation; Dashboard not accessible prior to verification.	PASS
4	Incorrect Password Validation: Enter wrong password on App Lock screen.	Argon2i verification returns FALSE; red error message 'Incorrect password!' displayed; application remains locked.	Red error feedback displayed; application remained locked.	PASS
5	Correct Password Validation: Enter registration password on App Lock screen.	Argon2i verification returns TRUE; application unlocks and navigates to Dashboard.	Successful unlock; navigation to Dashboard confirmed.	PASS
6	Logout – Session and Data Clearance: Press 'Log Out' on Profile screen.	User redirected to Login screen; DataStore fully cleared (all keys reset to null/false).	Navigation to authentication screen confirmed; Logcat shows DataStore clearance complete.	PASS

All six test scenarios (100%) returned a PASS status in the final iteration. The communication pipeline between the Jetpack Compose UI layer, the Argon2i cryptographic engine, and the Jetpack DataStore local storage operates correctly and consistently with the system design specification. This result provides empirical evidence that the Dual-Layer Security architecture is free of the previously identified defects and that Argon2i is successfully integrated at all three authentication checkpoints: Registration, Login, and App Lock verification.

Analysis of Findings

The testing results substantiate the research objectives on two levels: architectural correctness and cryptographic efficacy. From an architectural perspective, the MVVM + Dual-Layer Security design proved both functional and resilient. The strict separation between the ViewModel (cryptographic logic) and the View (Jetpack Compose UI) was instrumental in

isolating and resolving Defect 2; the hashing pipeline could be augmented without restructuring the UI layer. The identification and resolution of the race condition (Defect 1) furthermore demonstrates a non-trivial integration challenge: the asynchronous nature of Jetpack DataStore, while beneficial for preventing UI blocking, requires deliberate state management design—specifically the null-sentinel pattern—to prevent security state from being evaluated before it is initialized. This finding constitutes a practical contribution for Android developers integrating persistent security state with reactive UI frameworks. From a cryptographic perspective, the Argon2i parameter configuration (m=65536, t=2, p=1) provides a quantitatively strong defense against the primary threat vectors identified in the background. Specifically, the 64 MB per-attempt memory allocation renders GPU-based parallelized attacks economically infeasible: a high-end GPU with, for example, 12 GB of VRAM can theoretically support at most 192 concurrent Argon2i instances (12,288 MB / 64 MB), compared to the billions of concurrent hash attempts achievable with MD5 or SHA-256. The two-pass TMTO-resistant design further ensures that any attempt to reduce memory usage by substituting pre-computation for runtime memory access is thwarted. This directly addresses the vulnerability landscape documented[5][4].

The data-independent memory access property of Argon2i is particularly critical in the mobile context. On a shared or stolen device, a sophisticated adversary with physical access could potentially monitor memory access patterns of a running application to infer information about the password being processed. Argon2i's PRNG-driven, password-independent memory traversal eliminates this attack surface, as argued theoretically by the algorithm specification [9] and applicable to the App Lock scenario analyzed in this research. The PHC String Format storage approach further enhances long-term security maintainability. Because the hash string self-encodes the algorithm parameters (m, t, p, v), the verification function is parameter-agnostic; if future security research recommends increasing m or t, the system can transparently handle both old and new hash formats during verification, enabling a seamless security upgrade path.

Comparison with Related Studies

Table 11 provides a structured comparative analysis of this research against the five most closely related prior studies.

Table 11. Comparative Analysis with Related Studies

Study	Platform	Algorithm	Key Limitation	Advancement in This Study
Suendri [8] (2019)	Web (PHP)	Argon2 (web)	Server-side only; no mobile client-side implementation.	Native Android (client-side) Argon2i implementation with MVVM architecture.
Putra et al. [2] (2022)	Android	None (no hashing)	No cryptographic protection for local user data.	Full Argon2i hashing of credentials at all authentication checkpoints.
Alrizal [3] (2026)	Android	Weighted Scoring (gamification, not security)	No hashing layer or App Lock security mechanism.	Argon2i Dual-Layer Security integrated with existing gamification system.
Jehian et al. [10] (2025)	Web	Argon2 + ChaCha20-Poly1305	Web-based document encryption; no mobile App Lock gate.	Client-side App Lock using Argon2i on Android, protecting the application entry point.
Yolandari et al. [4] (2026)	Web (simulation)	Argon2 vs. Scrypt (comparison)	Simulation only; not implemented on real mobile architecture.	Full production implementation on Android with Black Box Testing validation.

This comparative analysis confirms the novelty of the present research. No prior study has accomplished the simultaneous integration of: (1) native Android Argon2i implementation, (2) client-side Dual-Layer Security combining Firebase Authentication and App Lock, (3) MVVM architectural pattern with Jetpack Compose, (4) Jetpack DataStore for asynchronous credential persistence, and (5) empirical validation through two-iteration Black Box Testing on a gamification-based productivity platform.

Limitations and Future Work

While the system achieves its stated objectives, several limitations were identified during the research process that constrain the current scope and define directions for future investigation. First, the evaluation was scoped exclusively to Black Box functional testing. Hardware profiling metrics—including precise CPU utilization percentage, dynamic RAM

consumption during Argon2i execution, and battery drain impact—were not measured. This is a consequential limitation, particularly for low-end Android devices (entry-level smartphones with limited RAM and slower processors), where the 64 MB memory allocation may impose non-trivial latency or thermal constraints. Future research should conduct systematic Hardware Profiling across a stratified sample of Android device specifications to establish performance bounds and potentially adapt the m and t parameters dynamically based on device capability.

Second, the system operates in an offline-first mode with no cloud backup of the Argon2i credential hash. If the user performs a logout, clears application data (via Android Settings), or switches devices, the local credential hash is permanently lost, necessitating re-registration. This is a significant usability limitation. Future development should investigate End-to-End Encrypted (E2EE) cloud synchronization of the Argon2i hash and salt, potentially using a key derived from the user's Firebase Authentication token as the encryption key for the cloud-stored local credential bundle.

Third, the current verification layer relies exclusively on Argon2i password entry. The integration of Android's native BiometricPrompt API supporting fingerprint and face authentication—as a faster secondary verification method would substantially improve user experience without compromising the Argon2i-based security foundation. The biometric layer would serve as a convenient alias for the underlying cryptographic verification, unlocking the DataStore credential without exposing the plaintext password.

CONCLUSION

This research successfully achieved both objectives: a Dual-Layer Security architecture was designed and implemented on an Android habit tracker application, integrating Firebase Authentication as the cloud-based primary layer and an Argon2i-powered App Lock as the client-side secondary layer, built on MVVM with Jetpack Compose and persistent, offline-first credential management via Jetpack DataStore; and the Argon2i algorithm was successfully implemented natively for password hashing across Registration, Login, and App Lock verification, using parameters m=65536, t=2, and p=1 to convert plaintext passwords into 256-bit hashes with a unique salt, stored in PHC String Format. Two-iteration Black Box Testing confirmed this success empirically: three critical defects identified in Iteration 1—including a UI race condition and an incomplete login hashing pipeline—were fully resolved by Iteration 2, achieving a 100% pass rate across all six test scenarios and verifying correct hashing, login/App Lock verification, App Lock enforcement on cold start and resume, accurate error feedback, and complete credential purging on logout. The main contribution is a validated technical blueprint for Argon2i integration in Android apps as a reference standard for client-side account security, alongside the identification and resolution of a common DataStore race-condition anti-pattern; future work includes hardware profiling for adaptive parameter tuning, BiometricPrompt integration, and E2EE cloud synchronization of credential data to remove the current device-lock limitation.

ACKNOWLEDGMENT

The author gratefully acknowledges the guidance and supervision of Khairuddin Nasution, S.T., M.Kom and Rachmat Aulia, S.Kom, M.Sc.IT throughout the research process, and expresses appreciation to the Department of Informatics Engineering, Faculty of Engineering, Universitas Islam Sumatera Utara, for providing the institutional support necessary to complete this work.

REFERENCES

Journal Article from the Internet

[1] S. Kemp, "Digital 2024: Indonesia," We Are Social & Meltwater, 2024. [Online]. Available: https://datareportal.com/reports/digital-2024-indonesia

[2] Putra et al., "Pengembangan Aplikasi Habit Tracker untuk Peningkatan Produktivitas Mahasiswa," Jurnal Sistem Informasi Akademik, 2022.

[3] A. Alrizal, "Pengembangan Aplikasi Habit Tracker Berbasis Android dengan Algoritma Weighted Scoring," Undergraduate Thesis, Universitas Islam Sumatera Utara, 2026.

[4] Yolandari et al., "Simulasi dan Perbandingan Hashing Password antara Algoritma Argon2 dan Scrypt," Jurnal Rekayasa Perangkat Lunak, 2026.

[5] J. Tippe and A. Berner, "Analisis Serangan Berbasis GPU dan ASIC terhadap Algoritma Hashing Tradisional," Jurnal Keamanan Sistem Informasi, 2025.
[6] BSSN, "Laporan Tahunan Keamanan Siber Indonesia 2024," Badan Siber dan Sandi Negara, 2024.
[7] Verizon, "Data Breach Investigations Report (DBIR) 2023," Verizon Enterprise, 2023. [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/
[8] Suendri, "Hashing Argon2 Untuk Keamanan Password Pada Sistem Berbasis Web Menggunakan PHP," JISTech (Journal of Islamic Science and Technology), vol. 4, no. 1, pp. 46–56, 2019.
[9] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications," in Proc. IEEE European Symposium on Security and Privacy (EuroS&P), 2017, pp. 292–302.
[10] Ichsan, A., Zulherry, A., Lubis, T. A., & Shahnaz, B. A. Z. (2025). Utilization of Mobile Applications to Speed Up The Search for Android-Based Index Places. IJATCoS: Indonesian Journal of Applied Technology. Computer and Science, 2(1).
[11] N. T. Jehian et al., "Keamanan Sistem Login Menggunakan Multifactor Authentication dan Algoritma Hashing," Journal of Information System, Informatics and Computing (JISICOM), vol. 9, no. 2, 2025.
[12] A. Muliawan and S. Hasnawati, "Penguatan Mekanisme Autentikasi Aplikasi Mobile sebagai Lini Pertahanan Data," Jurnal Teknologi Informasi, vol. 10, no. 2, 2024.
[13] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 9th ed. New York: McGraw-Hill Education, 2020.
[14] Zulherry, A., Basri, M., & Yusnandar, W. (2025). Implementation of Private Cloud Infrastructure as a Service Using Proxmox VE Based on the Network Development Life Cycle (NDLC) Approach. Altafani: Jurnal Pengabdian Masyarakat, 1(1).
[15] Ahlansyah, W., & Sari, I.P. (2026). Perancangan Tempat Sampah Pintar Berbasis IOT untuk Pemilahan Sampah Otomatis Menggunakan Sensor Infrared dan Pemantauan Real-Time. Hello World Jurnal Ilmu Komputer 5 (1), 59-68
[16] Zulherry, A., Sari, I. P., & Basri, M. (2026). Perancangan dan Implementasi Segmentasi LAN pada Infrastruktur Jaringan Skala Menengah. Hello World Jurnal Ilmu Komputer, 4(4).
[17] Google Developers, "Android Developers Documentation: Kotlin and Jetpack Compose," 2023. [Online]. Available: https://developer.android.com
[18] Sari, I.P., Bsri, M., & Syafrayani. 92025). Implementasi Sistem Aplikasi Pengolahan Teks pada Gambar Menggunakan Modifikasi Metode LSB dan ROT13. Hello World Jurnal Ilmu Komputer 4 (2), 90-97
[19] D. Jemerov and S. Isakova, Kotlin in Action. Shelter Island, NY: Manning Publications, 2017.
[20] W. Kurniawan, I. Prihandi, and N. Husufa, "Prototype Firebase Authentication Menggunakan Fitur Firebase Pada Aplikasi Android," Jurnal Satya Informatika, vol. 4, no. 1, pp. 71–78, 2019.
[21] H. Y. P. Napitupulu and I. G. D. Nugraha, "Sistem Berbasis Komputasi Kabut Untuk Sistem Parkir Pintar Terdesentralisasi Menggunakan Firebase," Jurnal Nasional Teknik Elektro dan Teknologi Informasi (JNTETI), vol. 13, no. 1, 2024.
[22] Sari, I.P., Hasibuan, A.R., & Hariani, P.P. (2025). TEXT PROCESSING APPLICATION ON IMAGES USING MODIFICATION METHOD LSB AND ROT13. Proceeding International Seminar on Islamic Studies 6 (1), 630-638
[23] C. Hanifurohman and D. D. Hutagalung, "Analisis Statis Menggunakan Mobile Security Framework untuk Pengujian Keamanan Aplikasi Mobile E-Commerce Berbasis Android," SEBATIK, vol. 25, no. 1, pp. 22–29, 2021.
[24] A. Caniago and T. Sutabri, "Analisis Dampak Psikologis dan Finansial Akibat Kebocoran Data Pribadi," Jurnal Keamanan Siber, vol. 5, no. 2, 2023.
[25] R. Octavia et al., "Analisis Kerentanan Autentikasi pada Aplikasi Mobile," Jurnal Keamanan Siber, 2024.
[26] Sari, I.P., & Manurung, A.A. (2025). Development of A Smart Monitoring System for IoT–Based Tide Observation. Al'adzkiya International of Computer Science and Information Technology (AIoCSIT) Journal, 6(2), 50-55
[27] Putra et al., "Dampak Finansial dari Pengambilalihan Akun Digital," Jurnal Keuangan dan Siber, 2024.
[28] R. Hutagaol et al., "Kesadaran Keamanan Siber pada Pengguna Smartphone di Indonesia," Jurnal Komputer dan Keamanan, vol. 11, no. 1, 2024.
[29] Zulherry, A., & Gunawan, M. (2025). Development of an Android-Based Smart Health Monitoring Device for Heartbeat Detection. Al'adzkiya International of Computer Science and Information Technology (AIoCSIT) Journal, 6(2), 75-79.
[30] J. Wetzels, "Open Source Password Hashing: Argon2," Security Research Paper, 2015.

[31] Y. Amrozi et al., "Tantangan Security dan Kehandalan Sistem dalam Aplikasi Bergerak," Jurnal Pendidikan Teknologi Informasi (JUKANTI), vol. 4, no. 2, pp. 1–10, 2021.
[32] Dian et al., "Implementasi Jetpack Compose pada Antarmuka Pengguna Aplikasi Android," Jurnal Teknologi Mobile, vol. 8, no. 3, 2022.