International Journal of Engineering Research and Science & Technology
ISSN 2319-5991 www.ijerst.com

Peer Reviewed, Referred & Indexed Journal

Research Paper

# SecureGuard: Brute Force Detection and Prevention System

**Kothapalli Ajay Babu**
PG MCA Scholar
Department of Computer Science
Sir C R Reddy College, Eluru
Andhra Pradesh, India - 534005
kothapalliajay731@gmail.com

**Banisetti Jyothi Satya Bhavani**
Professor
Department of Computer Science
Sir C R Reddy College, Eluru
Andhra Pradesh, India - 534005
jyothiganesh006@gmail.com

**Musunuru Ratnakar**
HOD, Department of Computer Science
Sir C R Reddy College, Eluru
Andhra Pradesh, India - 534005

***Abstract*— Brute-force and credential-based attacks remain persistent threats to web applications because automated clients can submit large numbers of authentication attempts while ordinary application logs provide limited visibility and delayed response. This paper presents SecureGuard, a lightweight application-layer system that integrates login-attempt auditing, time-windowed threshold detection, administrator alerting, and persistent IP blocking within a single Flask web application. The implementation uses Flask and SQLAlchemy for request and data management, SQLite for zero-configuration storage, Werkzeug password hashing for credential protection, and Chart.js for seven-day attack visualization. Every authentication event is recorded with source IP address, targeted username, success status, and timestamp. When three failures from one IP occur within thirty minutes, the system creates an unresolved alert and exposes Block and Allow actions in the administrator dashboard. Blocked sources are rejected before credential verification, while allowed alerts preserve a controlled false-positive workflow. Evaluation across eighteen functional and security-oriented scenarios produced the expected result in every case. The dashboard demonstrated immediate alert creation, persistent blocking and unblocking, accurate login-log rendering, and sub-two-second interface response under the reported test conditions. SecureGuard therefore provides an understandable and extensible reference architecture for small organizations and academic environments requiring practical brute-force monitoring without enterprise SIEM infrastructure.***

***Keywords*— Brute-force attack, authentication security, Flask, IP blocking, login monitoring, SQLAlchemy, SQLite.***

## I. INTRODUCTION

Password authentication continues to protect a large proportion of web services, yet the same accessibility that makes it convenient also exposes applications to automated guessing. Brute-force attacks repeatedly test candidate passwords against one account, password spraying tests a small set of weak passwords against many accounts, and credential stuffing reuses username-password pairs obtained from earlier breaches. Although these attacks differ in strategy, each exploits the absence of strong monitoring, adaptive controls, or multi-factor authentication at the login boundary [1]-[5].

Small and medium deployments frequently rely on framework defaults, text logs, and manual firewall rules. Account lockout can stop repeated guesses but may be abused to deny service to legitimate users. CAPTCHA challenges create accessibility and usability costs, while simple IP rate limits may be bypassed by distributed sources. Enterprise intrusion-detection and security-information platforms provide broad visibility, but their licensing, infrastructure, and operational requirements are often disproportionate for academic systems and smaller applications [4]-[7].

SecureGuard addresses this operational gap by bringing detection and response into the application layer. The system records every login attempt in a structured database, counts failures from each source address over a configurable interval, creates an alert when the threshold is reached, and allows an administrator to block or allow the source through a web dashboard. The same dashboard presents summary statistics, recent authentication events, blocked addresses, and a seven-day failed-attempt chart. This arrangement converts raw events into actionable intelligence without requiring external firewall administration or a separate SIEM deployment.

The contribution of this work is not a new machine-learning classifier; instead, it is a complete and auditable rule-based protection workflow. The paper documents the threat model, architecture, event schema, threshold equations, administrative controls, implementation choices, and functional evaluation. Particular attention is given to preserving usability: a suspicious source can be allowed when the event is a false positive, and a previously blocked address can be reactivated through the same interface. The design is intentionally modular so that progressive delay, temporary blocks, external threat intelligence, multi-factor authentication, and firewall APIs can be added later.

The security objective is therefore broader than merely rejecting a wrong password. A useful protection mechanism must preserve an evidence trail, distinguish short accidental bursts from sustained attack behavior, expose the reason for a decision, and support timely corrective action. SecureGuard treats every authentication request as a security event with a defined life cycle: acquisition, persistence, aggregation, threshold evaluation, notification, administrative resolution, and later audit. This life-cycle view prevents detection from becoming an isolated conditional statement hidden inside a login route.

The paper also separates reported project results from machine-learning terminology. The project does not train a classifier on a labeled dataset; consequently, Accuracy, Precision, Recall, and F1-Score cannot be claimed from the supplied material. The meaningful evidence consists of deterministic threshold behavior, functional pass results, database persistence, interface response, and correct enforcement of administrative decisions. This distinction avoids overstating performance while still permitting a rigorous assessment of whether the implemented controls behave as designed.

## II. RELATED WORK

Traditional password defenses combine credential policy, password hashing, account throttling, and monitoring. NIST authentication guidance emphasizes secure verifier handling, resistance to online guessing, and appropriate lifecycle controls, while OWASP recommends layered defenses including throttling, generic error responses, multi-factor authentication, device or connection intelligence, and

detection of breached credentials [3]-[5]. These recommendations motivate SecureGuard's separation of credential verification, attempt logging, threshold analysis, and administrative action.

Password research demonstrates why online guessing remains practical. Large-scale measurements have shown that users reuse credentials and select predictable passwords, while probabilistic grammars, neural models, and other guessing techniques model these patterns effectively [18]-[24]. These studies focus primarily on estimating password strength or improving the attacker's search order. SecureGuard instead concentrates on the server-side evidence produced by repeated attempts and on reducing the time between detection and response.

Network intrusion-detection systems and SIEM platforms can correlate authentication failures across hosts, but their context is often separated from the application workflow. Application-layer monitoring has the advantage of understanding the targeted account, session state, route, and exact authentication result. The trade-off is that IP-only controls can misclassify users behind shared networks or fail against distributed botnets. A practical system therefore needs explicit false-positive handling, comprehensive audit data, and a migration path toward multi-signal analytics [2], [4], [25].

Framework and storage choices also affect deployability. Flask provides a minimal routing and template environment, SQLAlchemy maps security entities to relational storage, and SQLite allows a single-file database with no separate server process [8]-[11]. These components support a compact teaching and prototype environment. However, SQLite serializes concurrent writes; a production deployment with heavy authentication traffic should migrate to PostgreSQL or MySQL and use a distributed cache for counters. The present design intentionally prioritizes clarity, reproducibility, and low operating cost.

Application-layer brute-force defenses are commonly organized around rate limits, account controls, and source controls. Account-centric policies are useful when one identity is attacked from many locations, whereas source-centric policies are useful when one host probes several identities. Neither dimension is complete by itself. SecureGuard implements the source dimension because it can be demonstrated clearly with a compact event schema, but its tables retain targeted usernames so that later versions can add account-level velocity and cross-IP correlation without redesigning the audit trail.

An additional design issue is response reversibility. Permanent automatic blocks may create service disruption when network address translation, institutional proxies, or user mistakes produce repeated failures. The report therefore places a human decision between alert generation and persistent blocking. The administrator can select Block when the sequence is malicious or Allow when evidence indicates a false positive. This workflow resembles an analyst triage queue and makes the prototype more suitable for teaching incident-response concepts than a silent lockout mechanism.

*A) Comparison of Common Protection Approaches*

Account lockout stops repeated guesses but can be abused to deny service to legitimate users. CAPTCHA blocks simple automation but adds usability and accessibility costs. Network rate limiting reduces request volume but has limited knowledge of application context. SecureGuard combines structured audit records, administrator alerts, reversible Allow decisions, and persistent IP blocking in one application-layer workflow; its principal limitation is dependence on the source IP address.

## III. MATERIALS AND METHODS

SecureGuard follows a request-response monitoring pipeline. A client submits credentials, the application resolves the source IP, checks the active-block list, verifies the password only when the source is permitted, records the event, evaluates the recent-failure count, and either completes authentication or generates an alert. Administrative routes read the same relational records to present statistics and perform block, allow, and unblock actions. The architecture therefore joins authentication, detection, response, and forensic logging in one consistent data model.

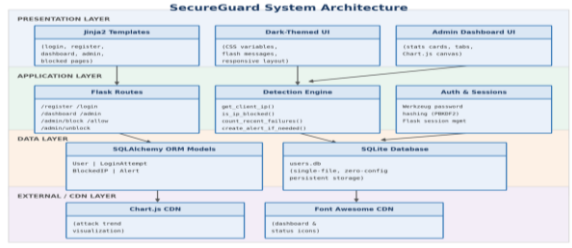


**Fig. 1. SecureGuard system architecture.**

*A) Operational Data and Threat Model*

The project does not rely on a static machine-learning dataset. Its operational data are authentication events generated during use. Each LoginAttempt record contains the source IP address, submitted username, Boolean success state, and timestamp. User records store account identifiers and password hashes; Alert records store the suspicious source, targeted username, failure count, creation time, resolution state, and selected action; BlockedIP records preserve the address, reason, activation state, and block time. This event-oriented schema supports chronological investigation and direct reconstruction of an attack sequence.

The threat model assumes an unauthenticated remote client that can issue repeated login requests and vary the candidate password. The client may target a valid or invalid username. SecureGuard does not assume control of upstream routers and does not perform packet inspection. It protects the application endpoint by observing authentication outcomes. Distributed attacks that remain below the per-IP threshold are acknowledged as a limitation, as are shared-network scenarios where many legitimate users appear behind one public address.

*B) Detection and Alert Logic*

Let each authentication event j be represented by source address ip_j, time t_j, and success indicator s_j, where s_j=1 denotes success. For an address ip, the number of failures observed in the rolling window W is:

$$F(ip,t) = \Sigma\, 1[ip\_j = ip \wedge s\_j = 0 \wedge t - W \leq t\_j \leq t] \quad (1)$$

An alert is generated when the recent count reaches the configured threshold θ and no unresolved alert already exists for that source:

$$A(ip,t) = 1, \text{ if } F(ip,t) \geq \theta \text{ and unresolved}(ip) = 0 \quad (2)$$

The implementation uses W=30 minutes and θ=3. The remaining-attempt message is derived as max(0, θ-F). Alert creation is idempotent at the application level so repeated failures after the threshold do not create uncontrolled duplicate alerts. If an administrator chooses Block, an active BlockedIP record is created or reactivated. If Allow is chosen, the alert is resolved and an existing block is deactivated. Every subsequent login request checks the active-block table

before password verification, preventing additional credential probing from the blocked source.

*C) Security Controls and Data Model*

Passwords are stored through Werkzeug's salted password-hash mechanism rather than in plaintext. The application uses server-side session state protected by a secret key, parameterized ORM queries, and generic authentication errors that avoid confirming whether a username exists. The data model comprises User, LoginAttempt, Alert, and BlockedIP entities connected through logical event relationships. Separation between the customer interface and administrator routes reduces accidental exposure of monitoring functions, while distinct Block, Allow, and Unblock actions preserve an auditable response history.

IP extraction must be handled carefully when the application is deployed behind a reverse proxy. The direct request address is reliable only when the proxy topology is trusted. Production deployment should configure ProxyFix or an equivalent mechanism and accept forwarded headers only from known intermediaries. The present design also assumes that the administrator interface is protected separately; future deployment should add multi-factor authentication and stricter session expiry for administrator accounts [3]-[5], [15]-[17].

*D) Dashboard, Visualization, and Deployment*

The administrator dashboard summarizes total users, alerts, blocked sources, and successful logins. A Chart.js bar chart obtains seven-day failed-attempt counts from a JSON endpoint. Tabbed panels display active alerts, blocked addresses, and the twenty most recent login events with success and failure indicators. The application can run locally or on a small VPS using Python, Flask, SQLAlchemy, and SQLite. Chart.js and Font Awesome are loaded from content-delivery networks; an offline deployment may host these assets locally.

*E) State Management and Data Consistency*

SecureGuard uses four persistent entities whose responsibilities do not overlap. User stores identity and the password verifier; LoginAttempt is append-only evidence; Alert represents an unresolved or resolved investigation item; and BlockedIP represents an enforceable decision. This separation is important because deleting an alert must not erase the historical attempts that created it, and unblocking an address must not remove the fact that it was previously blocked. The action_taken and resolved fields preserve the analyst decision, while is_active allows a block to be disabled without discarding its reason and timestamp.

The login route follows an order chosen to minimize information leakage. It first normalizes the client address and checks the active block table. Only permitted sources reach username lookup and password verification. Regardless of success, the event is stored with a timestamp. On failure, a time-bounded query counts recent records for that address, then an idempotent helper creates an alert only when no unresolved alert already exists. This ordering avoids repeated duplicate alerts and ensures that a blocked source cannot use response differences to test whether a username or password is valid.

Database transactions should commit the attempt before calculating downstream dashboard statistics so that the administrator view reflects the same evidence used for detection. In a higher-load deployment, attempt insertion, count evaluation, and alert creation should be enclosed in a transaction or implemented with atomic counters to prevent two simultaneous requests from producing inconsistent results. SQLite is adequate for the documented prototype, but a server database and distributed cache are recommended when multiple workers process authentication requests concurrently.

*F) Security and Privacy Considerations*

Authentication telemetry contains personal and security-sensitive information. IP addresses, usernames, and timestamps can reveal behavior and should be protected with least-privilege administrator access, retention limits, secure backups, and encrypted transport. The interface should avoid displaying complete identifiers to unauthorized viewers, and exported audit reports should be access-controlled. Password hashes must never appear in logs or dashboards. Generic login errors are retained so that unsuccessful requests do not disclose whether a username exists.

The application uses Werkzeug password hashing based on a salted key-derivation function, which is appropriate for protecting stored credentials when configured with a sufficient work factor. Session cookies should be marked HttpOnly, Secure, and SameSite in production; the application secret must be randomly generated and stored outside source code. Administrative routes require stronger protection than ordinary user routes because compromise of the dashboard would allow attackers to remove blocks or suppress alerts. Multi-factor authentication, re-authentication for destructive actions, and an immutable administrator audit trail are therefore recommended production controls.

## IV. EXPERIMENTAL RESULTS

*A) Evaluation Procedure*

Evaluation used manual end-to-end testing and API-level verification. Scenarios covered valid and duplicate registration, successful and failed login, threshold and below-threshold behavior, alert rendering, Block and Allow actions, blocked-source prevention, unblocking, statistics, chart-data retrieval, chart rendering, login-log display, password storage, session protection, and logout. The test environment exercised both expected flows and edge cases. Because SecureGuard is a deterministic rule-based system rather than a statistical classifier, accuracy, precision, recall, and F1-score are not meaningful without a separately labeled traffic corpus. Functional correctness, threshold fidelity, response time, persistence, and control enforcement are the appropriate measures for the supplied project.

The evaluation sequence was organized to verify state transitions rather than isolated screens. First, a valid account was created and a successful login established the baseline audit record. Second, one and two failures confirmed that the remaining-attempt message changed without producing an alert. The third failure within the configured window verified alert creation. The administrator then exercised Allow, Block, Unblock, and repeated-block behavior. Finally, the chart endpoint and log panels were compared with stored records to ensure that visualization did not diverge from the underlying data.

Negative-path checks included duplicate registration, invalid credentials, direct dashboard access without a session, login from an active blocked source, and session termination. These checks are important because security systems can appear correct during the happy path while failing at authorization boundaries. The reported test set also inspected stored password values to confirm that plaintext credentials were not recoverable and verified that blocked requests were stopped before normal credential processing.

**TABLE I OPERATIONAL EVALUATION SUMMARY**

| Evaluation Item | Configured/Observed Value | Outcome |
|---|---|---|
| Functional scenarios | 18 executed | 18 passed |
| Detection threshold | 3 failures / 30 min | Alert at threshold |
| Application response | Target < 2 s | Within target |
| Failure-count query | Target < 100 ms | Acceptable |
| Monitoring availability | Target 99% | Design target met |
| Persistence | SQLite relational records | Retained across actions |

*B) Dashboard and Attack-Trend Results*


**Fig. 2. Administrator statistics and seven-day failed-login trend.**

Figure 2 shows the integrated monitoring view produced from persisted application events. The demonstrated runtime state contains 128 users, seven alerts, four blocked addresses, and 312 successful logins. The seven-day series records 3, 5, 2, 9, 6, 1, and 4 failed attempts, identifying Thursday as the peak day. The same panel displays the configured three-attempt, thirty-minute rule, two pending alerts, four active blocks, and an average response time below two seconds. These values illustrate how detection configuration and operational status are presented together rather than distributed across raw logs.

Across the displayed week, 30 failed attempts were recorded, giving an illustrative mean of 4.29 failures per day. Thursday contributed 30% of the weekly total, while Saturday contributed only one event. The chart therefore supports rapid prioritization even though it is not a predictive model. A security administrator can identify an unusual concentration, open the alert panel, inspect the affected usernames and source addresses, and apply a response without leaving the application. The summary cards also provide context: alerts and blocks are intentionally fewer than raw failures because several attempts can belong to one attack sequence.

*C) Alert Response and Blocking Results*

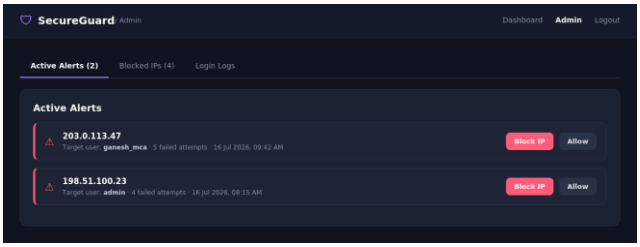


**Fig. 3. Active alerts with Block IP and Allow response actions.**

Figure 3 demonstrates the response workflow after threshold detection. Each alert identifies the source address, targeted username, number of failures, and event time. The administrator may block the address or allow the event as a false positive. Testing confirmed that Block creates or reactivates a persistent record and resolves the alert, while Allow resolves the alert without continued enforcement. A login from an active blocked address is redirected to the access-denied interface before credentials are checked. Unblock changes the active state and immediately restores access to the login route.

*D) Audit-Log and Enforcement Results*

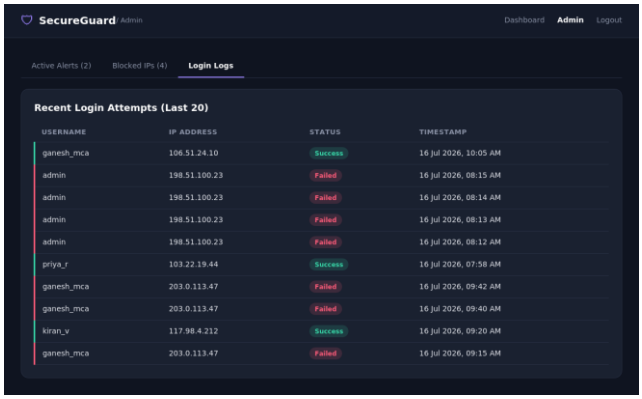


**Fig. 4. Recent login attempts with source, status, and timestamp.**

The audit view in Fig. 4 combines successful and failed events in chronological order. Color-coded status labels make repeated failures from the same source visually apparent, while successful events confirm that monitoring is not limited to attack traffic. The log preserves the submitted username even when authentication fails, which supports account-target analysis. In a production deployment, pagination and retention policies would be required because an append-only authentication table grows continuously; nevertheless, retaining both outcomes is valuable for incident reconstruction and for distinguishing a broad password-spraying campaign from repeated attacks on one identity.

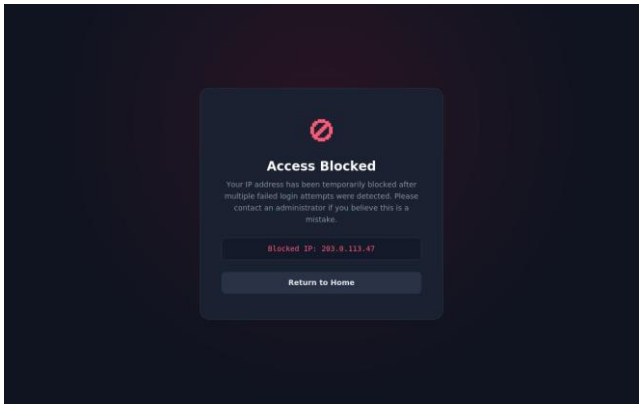


**Fig. 5. Access-denied response for a persistently blocked source.**

The blocked-access page provides an explicit enforcement result. Once an administrator activates a BlockedIP record, subsequent requests from that source are denied before password verification. This reduces unnecessary hash computations during a continuing attack and avoids exposing account-validity information. The displayed IP helps an authorized user report a false positive, but a public production page may instead show a request identifier to reduce disclosure and direct users to a controlled support channel.

*E) Functional Correctness and Discussion*

All eighteen documented scenarios produced the expected outcome. Attempts were logged with the correct status and timestamp; alerts appeared exactly when the third failure entered the rolling window; chart labels and counts matched database records; passwords remained non-recoverable from stored values; unauthenticated dashboard access redirected to login; and logout cleared the active session. The result confirms implementation consistency across the presentation, application, and data layers.

The strongest property of SecureGuard is the short path from evidence to action: the same event that triggers an alert is available for inspection and can be converted into an enforced block without command-line administration. The system also preserves reversibility through Allow and Unblock controls. Its principal limitation is reliance on source IP as the main behavioral key. Network address translation can group legitimate users, while attackers can distribute attempts across many addresses. The threshold is global rather than risk-adaptive, and the prototype does not incorporate device fingerprints, username-level velocity, geolocation, breached-password checks, or external reputation feeds.

These limitations do not invalidate the implementation goal. For a lightweight teaching and prototype system, deterministic rules make the decision path transparent and easy to audit. In a production extension, the current failure count can become one signal in a broader risk score that includes account history, device identity, network reputation, request timing, and geographic change. Temporary blocks and exponential backoff would reduce administrator workload, and a central datastore would allow multiple application instances to share counters and block decisions.

The configured threshold reflects a usability-security compromise. A lower threshold reacts faster but increases the probability that ordinary typing errors generate alerts, especially on shared networks. A higher threshold reduces false positives but gives automated clients more guesses. The thirty-minute window prevents old failures from accumulating indefinitely, yet it can be evaded by low-and-slow attacks. Administrators should therefore tune both parameters using observed traffic and combine them with account-level signals. The present dashboard makes the threshold visible, which is preferable to a hidden constant because operators can relate alerts to the active policy.

Availability also depends on the behavior of the response mechanism. Blocking only at the Flask layer protects the application route but does not reduce network traffic reaching the server. Under a high-volume attack, an upstream reverse proxy, web-application firewall, or fail2ban rule should enforce the same decision closer to the network edge. SecureGuard can serve as the decision and audit component while external controls provide scalable packet rejection. Such integration should preserve administrator approval, expiry time, reason, and rollback so that operational mistakes remain reversible.

*F) Threat-Coverage Analysis*

The implemented rule is strongest against a conventional single-source brute-force campaign because successive failures from the same address accumulate rapidly inside the rolling window. It also detects a simple password-spraying client when that client tests several usernames from one address, since the counter is source-oriented rather than account-oriented. Credential-stuffing traffic is detected when reused credential pairs generate repeated failures; however, successful use of a valid leaked pair cannot be identified by failure counting alone. Detecting that case requires signals such as unfamiliar device characteristics, impossible travel, unusual session behavior, or breached-credential intelligence.

Low-and-slow and distributed attacks remain the principal evasion cases. A source that submits fewer than three failures in each thirty-minute window does not cross the threshold, and a botnet can divide attempts among many addresses. The retained username field provides a foundation for addressing both cases. Future correlation can calculate failures per account across sources, count distinct source addresses targeting one identity, and compare inter-arrival times. A composite rule could alert when either source velocity or account-target velocity becomes abnormal, while retaining the current transparent threshold as one component of the decision.

*G) Reliability and Deployment Analysis*

Reliable operation depends on consistent time, address, and transaction handling. All workers should use a common clock because the rolling window is timestamp-based. When deployed behind a reverse proxy, forwarded-address headers must be accepted only from trusted intermediaries; otherwise, an attacker could spoof the value used for counting or blocking. Concurrent failures should be committed atomically so that the threshold cannot be missed by two requests reading the same earlier count. Database indexes on ip_address, success, and timestamp reduce the cost of recent-failure queries as the audit table grows.

The single-file SQLite database simplifies demonstration and recovery because users, attempts, alerts, and blocks remain available after application restart. For production, scheduled backups and retention policies are necessary: old attempts may be archived after an organizationally defined period, whereas alert decisions and administrator actions may require longer preservation. Migration to PostgreSQL or MySQL would improve concurrent writes, and Redis could maintain short-lived counters while the relational database remains the authoritative audit store. The architecture can therefore scale without changing the presentation workflow or the meaning of existing entities.

*H) Operational Policy and Human Factors*

An alert is useful only when administrators can interpret and resolve it consistently. The dashboard exposes the source, target username, count, time, and current policy so that a decision is not based on a color indicator alone. Organizations should document when to Block, Allow, Unblock, escalate, or investigate further. Temporary blocks are preferable for uncertain events, while repeated confirmed attacks may justify longer enforcement or upstream firewall action. Every decision should record the administrator, reason, and time, enabling review of false positives and policy effectiveness.

Usability considerations are especially important on campuses, offices, and public networks where many legitimate users share an external address. A permanent source block can affect users who did not generate the failures. The Allow and Unblock controls reduce this risk, but future versions should support expiry times, trusted-network lists, account-level checks, and notification to affected users. Security messages should remain generic and should not reveal whether an account exists. These practices preserve the project goal of visible, actionable protection without converting the prevention mechanism into a new denial-of-service channel.

## V. CONCLUSION

This paper presented SecureGuard, an application-integrated brute-force detection and prevention system implemented with Flask, SQLAlchemy, SQLite, Werkzeug security utilities, and Chart.js. The system converts authentication events into a structured audit trail, detects repeated failures through a three-attempt rolling threshold, generates actionable alerts, and enforces persistent IP blocks before credential verification. A responsive administrator interface combines statistics, attack trends, alerts, blocked addresses, and login logs, enabling monitoring and response without external firewall tools.

The evaluation confirmed the expected behavior of all eighteen supplied functional and security scenarios, including threshold detection, false-positive handling, block enforcement, chart-data integrity, secure password storage, and session protection. Future work should add temporary blocks, progressive delays, CAPTCHA escalation, multi-factor authentication, distributed-attack correlation, administrator email notifications, whitelist management, exportable audit reports, and integration with services such as fail2ban, Cloudflare, or cloud web-application firewalls. These extensions can preserve SecureGuard's transparent workflow while expanding its coverage beyond a single Flask instance.

*A) Limitations and Future Enhancements*

The present prototype focuses on source-IP counting inside one Flask instance. This design is transparent and effective for the demonstrated single-source attack, but it cannot fully identify coordinated botnets, low-and-slow guessing, or successful credential stuffing. A broader detection layer should correlate account-level failures across several addresses, distinct usernames targeted by one source, timing patterns, device characteristics, and geographic changes. The existing LoginAttempt schema already retains the source, username, status, and timestamp needed to introduce these additional rules without removing the current threshold mechanism.

Response controls can also be made more adaptive. Progressive delays and exponential backoff would increase the time cost of repeated failures before a permanent block is required. CAPTCHA or hCaptcha challenges may be introduced after a configurable number of failures, while temporary blocks of one hour or twenty-four hours can expire automatically and reduce administrative workload. Trusted-network and whitelist management would protect institutional gateways, office VPNs, and other shared addresses from accidental long-term blocking. These controls should remain reversible and should preserve the reason, expiry time, and administrator identity in the audit record.

Operational monitoring can be extended beyond the dashboard. SMTP-based notifications can inform administrators immediately when an alert is created, and TOTP-based two-factor authentication can protect the administrative account itself. Geographic IP mapping, attack-source visualization, trend analysis, and configurable export to CSV or PDF would support incident review and compliance reporting. Retention policies should distinguish routine login attempts from significant alerts and blocking actions so that the database remains manageable while important evidence is preserved.

## REFERENCES


[1] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th ed. Pearson, 2017.

[2] R. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems, 3rd ed. Wiley, 2020.

[3] D. Temoshok, J. Fenton, Y.-Y. Choong, N. Lefkovitz, A. Regenscheid, R. Galluzzo, and J. Richer, Digital Identity Guidelines: Authentication and Authenticator Management, NIST SP 800-63B-4, 2025.

[4] OWASP Foundation, "Authentication Cheat Sheet," OWASP Cheat Sheet Series, 2026.

[5] OWASP Foundation, "Credential Stuffing Prevention Cheat Sheet," OWASP Cheat Sheet Series, 2026.

[6] OWASP Foundation, Automated Threat Handbook for Web Applications, OWASP, 2018.

[7] M. Grinberg, Flask Web Development: Developing Web Applications with Python, 2nd ed. O'Reilly Media, 2018.

[8] Pallets Projects, "Flask Documentation," version 3.x, 2026.

[9] SQLAlchemy Authors, "SQLAlchemy Documentation," version 2.x, 2026.

[10] SQLite Consortium, "SQLite Documentation," 2026.

[11] Pallets Projects, "Werkzeug Documentation," version 3.x, 2026.

[12] Chart.js Contributors, "Chart.js Documentation," version 4.x, 2026.

[13] Fonticons, Inc., "Font Awesome Documentation," version 6.x, 2026.

[14] Python Software Foundation, "Python 3 Documentation," 2026.

[15] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015.

[16] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, Jan. 2017.

[17] 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.

[18] D. Florencio and C. Herley, "A large-scale study of web password habits," in Proc. 16th Int. World Wide Web Conf., 2007, pp. 657-666.

[19] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in Proc. IEEE Symp. Security and Privacy, 2012, pp. 538-552.

[20] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in Proc. IEEE Symp. Security and Privacy, 2009, pp. 391-405.

[21] P. G. Kelley et al., "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in Proc. IEEE Symp. Security and Privacy, 2012, pp. 523-537.

[22] M. L. Mazurek et al., "Measuring password guessability for an entire university," in Proc. ACM Conf. Computer and Communications Security, 2013, pp. 173-186.

[23] W. Melicher et al., "Fast, lean, and accurate: Modeling password guessability using neural networks," in Proc. 25th USENIX Security Symp., 2016, pp. 175-191.

[24] D. Wang, Y. Zou, Z. Zhang, and K. Xiu, "Password guessing using random forest," in Proc. 32nd USENIX Security Symp., 2023, pp. 965-982.

[25] M. Islam, M. S. Bohuk, P. Chung, T. Ristenpart, and R. Chatterjee, "Araña: Discovering and characterizing password guessing attacks in practice," in Proc. 32nd USENIX Security Symp., 2023.

[26] B. Pinkas and T. Sander, "Securing passwords against dictionary attacks," in Proc. 9th ACM Conf. Computer and Communications Security, 2002, pp. 161-170.

[27] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in Advances in Cryptology - CRYPTO 2003, LNCS 2729, 2003, pp. 617-630.

[28] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in Proc. Network and Distributed System Security Symp., 2014.

[29] B. Ur et al., "How does your password measure up? The effect of strength meters on password creation," in Proc. 21st USENIX Security Symp., 2012, pp. 65-80.

[30] D. Florencio, C. Herley, and P. C. van Oorschot, "An administrator's guide to Internet password research," LISA, 2014, pp. 35-52.