A Fault-Tolerant Cryptographic Key Recovery Framework Using Reed-Solomon-Decoded Threshold Secret Sharing

Nayaka Ghana Subrata – 13523090[1,2]
Department of Informatics Engineering
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Jl. Ganesha 10, Bandung 40132, Indonesia
[1]13523090@std.stei.itb.ac.id, [2]nayakaghana39@gmail.com

***Abstract*—Cryptographic systems depend on secret keys, yet storing a key in a single location creates a single point of failure. If the key is lost, encrypted data becomes unrecoverable; if the key is copied broadly, the risk of unauthorized disclosure increases. This paper implements and evaluates a fault-tolerant key recovery framework that combines Shamir Secret Sharing with Reed-Solomon-style Berlekamp-Welch decoding. An AES-256 key is first used to protect data with AES-GCM and is then divided into threshold shares over a prime finite field. During recovery, the reconstruction stage can tolerate missing shares and can also correct a limited number of corrupted shares when sufficient redundancy is available. A modular Python prototype is developed to support key generation, authenticated encryption, share creation, robust reconstruction, and experiment automation. Across fifteen experimental scenarios and 375 total trials, all observed outcomes matched the theoretical threshold and decoding bounds. Recovery succeeded in every scenario that satisfied the required redundancy condition and failed in every scenario that violated it. The results show that Reed-Solomon-decoded reconstruction improves availability and fault tolerance without weakening the threshold secrecy property provided by Shamir Secret Sharing.**

***Index Terms*—Shamir Secret Sharing, Reed-Solomon Codes, Berlekamp-Welch Decoding, AES Key Recovery, Threshold Cryptography, Robust Secret Sharing**

I. Introduction

The operational security of symmetric cryptography depends on secret keys. When an encryption key is stored in a single database, device, or password vault, the entire protection mechanism depends on the continued availability and integrity of that one location. The failure modes are immediate: key loss makes ciphertext permanently unreadable, while key compromise allows an attacker to decrypt all protected data. For this reason, practical key management must address both confidentiality and recoverability.

Threshold secret sharing offers a direct way to remove this single point of failure. Shamir's scheme divides a secret into multiple shares such that any threshold number of shares can reconstruct the secret, while any smaller set reveals no information about it [1]. The construction reduces the access-control problem to polynomial interpolation over a finite field. However, standard reconstruction assumes that the shares presented by participants are correct. In realistic settings this assumption is weak: shares may be unavailable, corrupted by storage faults, or intentionally modified by an adversary.

The close algebraic relationship between Shamir sharing and Reed-Solomon codes makes it natural to strengthen reconstruction with error correction [2]. Both mechanisms evaluate a low-degree polynomial at distinct field points. As a result, robust recovery can be formulated as recovering the original polynomial from noisy point evaluations, which is precisely the setting addressed by Reed-Solomon decoding algorithms. In particular, the Berlekamp-Welch perspective allows reconstruction to remain correct even when a bounded number of submitted shares are wrong [3].

This paper studies a practical key recovery prototype rather than a new cryptographic primitive. The protected secret is an AES-256 key [4], used with the AES-GCM authenticated encryption mode [5]. The contribution lies in implementing a modular end-to-end system and experimentally validating the threshold and error-correction behavior of the resulting recovery process.

The main contributions of this work are as follows:

1) A modular Python implementation that integrates AES-GCM encryption, Shamir share generation, and Reed-Solomon-style Berlekamp-Welch reconstruction into a single key recovery workflow.
2) An experimental evaluation over fifteen recovery scenarios that cover normal recovery, minimum-threshold recovery, insufficient-share failure, and recovery with corrupted shares.
3) A grounded analysis showing that robust decoding improves availability and fault tolerance without reducing the secrecy threshold of the original sharing scheme.

The remainder of the paper is organized as follows. Section II reviews the necessary theoretical background on AES-GCM, finite-field arithmetic, Shamir sharing, Reed-Solomon codes, and Berlekamp-Welch decoding. Section III presents the proposed recovery framework. Section IV describes the implementation. Section V defines the experiments, while Section VI reports the results and discusses their implications.

Section VII analyzes the security model and limitations. Section VIII concludes the paper.

II. Theoretical Background

A. AES-GCM

The Advanced Encryption Standard (AES) is a NIST standard block cipher with 128-bit block size and key sizes of 128, 192, and 256 bits [4]. This work uses AES-256 because the goal is not only to split a secret but also to protect a realistic high-value key. Encryption is performed with Galois/Counter Mode (GCM), which provides confidentiality together with integrity through authenticated encryption [5]. In the prototype, the data owner first encrypts a plaintext under AES-GCM and then protects the resulting AES key with the threshold recovery mechanism.

B. Finite-Field Arithmetic

Shamir sharing and Reed-Solomon decoding both operate over a finite field. Let $\mathbb{F}_p$ be a prime field where all additions, subtractions, multiplications, and inverses are taken modulo a prime p. The prototype uses the NIST P-384 prime field, which is larger than the 256-bit AES key space and therefore allows an AES-256 key to be embedded directly as a field element without fragmentation.

Given coefficients $a_0, a_1, \ldots, a_d \in \mathbb{F}_p$, a polynomial is written as

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d. \tag{1}$$

Polynomial evaluation and interpolation are efficient over $\mathbb{F}_p$, which makes the field suitable both for share generation and for robust reconstruction.

C. Shamir Secret Sharing

In a standard (t, n) Shamir scheme, the secret s is stored as the constant term of a random polynomial of degree $t-1$:

$$f(x) = s + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1}, \tag{2}$$

where $a_1, \ldots, a_{t-1}$ are chosen uniformly from $\mathbb{F}_p$ [1]. A share is the pair

$$(x_i, y_i) = (x_i, f(x_i)). \tag{3}$$

Any set of t valid shares uniquely determines $f(x)$, and therefore the secret is recovered by evaluating

$$s = f(0). \tag{4}$$

If fewer than t shares are known, the missing coefficients remain unconstrained, so the secret remains information-theoretically hidden [1].

D. Lagrange Reconstruction

When no share corruption is present, recovery uses interpolation. Given t valid shares $(x_1, y_1), \ldots, (x_t, y_t)$, the polynomial can be written in Lagrange form as

$$f(x) = \sum_{i=1}^{t} y_i \ell_i(x), \tag{5}$$

where

$$\ell_i(x) = \prod_{\substack{1 \le j \le t \\ j \ne i}} \frac{x - x_j}{x_i - x_j}. \tag{6}$$

Substituting $x = 0$ yields the secret directly:

$$s = f(0) = \sum_{i=1}^{t} y_i \ell_i(0). \tag{7}$$

This formula is sufficient when all submitted shares are valid, but it fails to provide protection against undetected errors in the y values. That limitation motivates the use of Reed-Solomon decoding during reconstruction.

E. Reed-Solomon Codes

Reed-Solomon codes also encode information by evaluating a low-degree polynomial at multiple field points [2]. This structural similarity is the key link used in this paper:

$$\text{Shamir sharing} \iff \text{secure polynomial sharing}, \tag{8}$$

$$\text{RS decoding} \iff \text{recovery from noisy shares}. \tag{9}$$

For a degree-$(t-1)$ polynomial evaluated at n points, the associated Reed-Solomon code has minimum distance

$$d_{\text{min}} = n - t + 1. \tag{10}$$

This quantity explains the asymmetry between missing and corrupted shares. Missing shares only reduce the number of equations, so recovery requires at least $m \ge t$ available shares. Corrupted shares are harder because the decoder must both identify errors and recover the polynomial, which requires extra redundancy. In coding-theoretic terms, unique decoding can tolerate up to $\lfloor (d_{\text{min}} - 1)/2 \rfloor$ symbol errors when all n evaluations are present.

F. Berlekamp-Welch Decoding

To recover from corrupted shares, this work adopts a Reed-Solomon-style Berlekamp-Welch approach [3]. Suppose a degree-$(t-1)$ polynomial is observed through m shares, of which at most e are corrupted. Correct recovery is possible when

$$m \ge t + 2e. \tag{11}$$

The decoder searches for an error-locator polynomial $E(x)$ of degree e and a combined polynomial $Q(x)$ such that

$$Q(x_i) = y_i E(x_i) \tag{12}$$

for all received shares (x_i, y_i). If a solution exists and $E(x)$ divides $Q(x)$, then the original secret polynomial is recovered as

$$f(x) = \frac{Q(x)}{E(x)}. \tag{13}$$

The role of $E(x)$ is to vanish at erroneous evaluation points, while $Q(x)$ absorbs the discrepancy between the correct polynomial values and the corrupted observations. In practice the decoder solves a linear system for the unknown coefficients of $Q(x)$ and the non-leading coefficients of $E(x)$, then verifies that the quotient has the expected degree. This verification step is important because it rejects inconsistent share sets instead of silently returning an arbitrary polynomial.

The Reed-Solomon component improves robustness rather than secrecy. It corrects a bounded number of wrong shares, but an adversary who gathers at least t valid shares can still reconstruct the secret by design. This separation between confidentiality and fault tolerance is central to the interpretation of the experimental results.

III. Proposed Key Recovery Framework

The proposed framework protects an AES-256 key through threshold sharing while extending the recovery phase with Reed-Solomon-style error correction. The complete workflow has eight stages:

1) Generate a random AES-256 key.
2) Encrypt the target plaintext with AES-GCM.
3) Convert the AES key to an integer in $\mathbb{F}_p$.
4) Split the integer secret into n Shamir shares with threshold t.
5) Distribute or store the shares across independent locations.
6) During recovery, collect m available shares.
7) Apply robust reconstruction to tolerate missing or corrupted shares.
8) Use the recovered key to decrypt the ciphertext and verify correctness.

The ordering of these steps matters. The plaintext is encrypted before the key is split so that the recovery procedure handles only a short fixed-size secret rather than an arbitrarily large message. This keeps the secret-sharing layer compact and ensures that the cost of robust reconstruction depends on the key-management problem instead of on the size of the protected data.

Fig. 1 illustrates the end-to-end workflow from AES key generation to robust share-based recovery.

Algorithm 1 summarizes the recovery logic. The key observation is that the framework does not replace Shamir's access policy; it refines only the reconstruction stage. If the available shares are all valid, robust recovery reduces to ordinary polynomial interpolation. If a bounded number of shares are corrupted and the redundancy condition in (11) holds, the Berlekamp-Welch step recovers the intended polynomial and therefore the original AES key.

Algorithm 1 Robust AES Key Recovery

Require: Threshold t, total shares n, available shares $\mathcal{S}$, AES-GCM ciphertext package C
Ensure: Recovered AES key K or failure
1: Generate a random AES-256 key K
2: Encrypt plaintext with AES-GCM under K to obtain C
3: Encode K as field element $s \in \mathbb{F}_p$
4: Create degree-$(t-1)$ Shamir polynomial $f(x)$ with constant term s
5: Distribute n shares $(x_i, f(x_i))$
6: Collect available shares $\mathcal{S}$ during recovery
7: **if** $|\mathcal{S}| < t$ **then**
8:     **return** failure
9: **end if**
10: Use Berlekamp-Welch decoding on $\mathcal{S}$ to recover $\hat{f}(x)$
11: Extract $\hat{s} = \hat{f}(0)$ and convert it to $\hat{K}$
12: Attempt AES-GCM decryption of C with $\hat{K}$
13: **if** decryption and authentication succeed **then**
14:     **return** $\hat{K}$
15: **else**
16:     **return** failure
17: **end if**

A. Design Assumptions and Parameter Selection

The framework assumes that the AES key is generated with cryptographically secure randomness and that shares are stored across distinct failure domains. In this setting the threshold t controls the balance between confidentiality and recoverability. A smaller threshold improves availability because fewer shareholders are required during recovery, but it also lowers the number of compromised shares needed for disclosure. A larger threshold has the opposite effect.

The total share count n controls the redundancy budget. When no corruption is expected, the basic reconstruction requirement is only $m \geq t$. When corrupted shares are possible, the choice of n should allow the intended recovery set to satisfy (11). For example, if the design target is to tolerate one corrupted share in a $(3, n)$ system while still recovering from all available shares, then n should be at least 5. If the target is to tolerate two corrupted shares in a $(5, n)$ system, then at least 9 shares must participate in reconstruction.

This design targets a specific threat model. It tolerates accidental share loss, partial share compromise below the threshold, and bounded corruption during recovery. It does not claim resistance against an attacker who already possesses at least t valid shares, nor does it protect a workstation that is compromised after the key has been reconstructed.

IV. Implementation

A. Software Architecture

The prototype is implemented in Python 3.12 and organized into modular components:

Fig. 1. End-to-end workflow of the proposed robust AES key recovery framework.

- `aes_utils.py`: AES-256 key generation, AES-GCM encryption and decryption, and conversion between byte strings and field elements.
- `field.py`: prime-field arithmetic, polynomial evaluation, and polynomial division helpers.
- `shamir.py`: random polynomial generation, share creation, ordinary secret reconstruction, share selection, and share corruption simulation.
- `robust_rs.py`: modular Gaussian elimination, Berlekamp-Welch decoding, and robust secret recovery.
- `experiments.py`: automated scenario execution and timing collection.

This split keeps cryptographic operations, algebraic utilities, and experiment control separate. The layout also makes it easier to replace the reconstruction method in future work, for example with verifiable or authenticated secret sharing variants.

B. Finite-Field and Serialization Choices

The implementation uses the NIST P-384 prime field in order to embed a 256-bit AES key directly into one field element. This avoids splitting the key into multiple blocks while still keeping the field size reasonably compact. Shares are serialized using fixed-width field encoding for both the x and y coordinates. As a result, each coordinate occupies 48 bytes and each share occupies 96 bytes in the current prototype. This decision simplifies the implementation and makes storage-overhead calculations deterministic, although a more compact representation for the x coordinate is possible.

C. Recovery Logic

The decoder accepts the set of available shares and automatically tries the largest feasible error budget first, namely $\lfloor (m-t)/2 \rfloor$, before falling back to smaller budgets. For a fixed error count e, the implementation solves the linear system induced by (12), reconstructs $Q(x)$ and $E(x)$, divides the polynomials, and verifies that the quotient has degree less than t. The reconstructed constant term is then converted back into a 32-byte AES key. As a final correctness check, AES-GCM decryption must also succeed, which ensures that an algebraically plausible but incorrect secret is rejected.

V. Experimental Design

A. Scenarios

The evaluation focuses on correctness, decryption success, failure behavior, timing, and storage overhead. Fifteen scenarios were selected to cover both the threshold property and the Reed-Solomon correction bound. Each scenario was executed for 25 independent trials, giving 375 total runs.

Table I lists the evaluated configurations. The "available" count m denotes the number of shares presented to the decoder, while e denotes the number of shares intentionally corrupted before recovery. Scenarios marked as failure are expected to fail either because $m < t$ or because the redundancy condition $m \geq t + 2e$ is not satisfied.

B. Measurement Procedure

Each trial follows the same procedure. First, a fresh AES-256 key is generated and used to encrypt a benchmark plaintext with AES-GCM. Second, the key is split into shares under the specified (t, n) configuration. Third, a random subset of m shares is selected. Fourth, the scenario optionally corrupts e of those shares by modifying their y values. Finally, the recovery stage attempts to reconstruct the key and decrypt the ciphertext. Success is recorded only when the recovered key exactly matches the original key and AES-GCM verification succeeds.

Timing is measured separately for share generation and for recovery. Storage overhead is computed as the total serialized share size divided by the original 32-byte AES key size. Because the serialization format is fixed-width, the overhead depends only on the number of generated shares.

C. Evaluation Metrics

Five metrics are used in the evaluation. First, *recovery success* records whether the robust decoder returns a candidate secret without violating the algebraic constraints. Second, *key correctness* checks whether the recovered 32-byte value matches the original AES key exactly. Third, *decryption success* checks whether AES-GCM accepts the recovered key and reproduces the original plaintext. Fourth, *recovery latency* measures the wall-clock time of the reconstruction stage. Fifth, *storage overhead* measures the total share footprint relative to the 32-byte secret.

These metrics distinguish between algebraic recovery and application-level recovery. In principle a decoder might output a field element even when it is not the original key. By checking both exact key equality and authenticated decryption, the evaluation rejects such cases. This distinction is important because a key-recovery system is useful only when the recovered secret is operationally valid for the protected ciphertext.

VI. Results and Discussion

A. Functional Verification

The first result is categorical: every scenario behaved exactly as predicted by theory. Across 375 total trials, all 275 trials in theoretically valid scenarios produced the correct AES key and successfully decrypted the ciphertext, while all 100

TABLE I
EXPERIMENTAL SCENARIOS AND OBSERVED OUTCOMES ACROSS 25 TRIALS PER SCENARIO

Scenario	t	n	m	e	Observed Success Rate	Mean Recovery (ms)
Normal recovery	3	5	5	0	100%	0.1419
Minimum valid recovery	3	5	3	0	100%	0.0736
Insufficient shares	3	5	2	0	0%	0.0017
One corrupted share, enough redundancy	3	5	5	1	100%	0.1781
One corrupted share, not enough redundancy	3	5	4	1	0%	0.0402
Larger normal recovery	4	7	7	0	100%	0.2937
Larger minimum recovery	4	7	4	0	100%	0.1375
Larger insufficient shares	4	7	3	0	0%	0.0018
Correct one corrupted share	4	7	6	1	100%	0.2397
Correct two corrupted shares	4	8	8	2	100%	0.4009
Large threshold normal	5	10	10	0	100%	0.6451
Large threshold minimum	5	10	5	0	100%	0.1254
Large threshold insufficient	5	10	4	0	0%	0.0018
Large threshold, one corrupted	5	10	7	1	100%	0.2841
Large threshold, two corrupted	5	10	9	2	100%	0.4787

trials in theoretically invalid scenarios failed during robust reconstruction. No case produced a false positive decryption.

This outcome directly matches the two central recovery rules:

- If there are no corrupted shares, recovery succeeds whenever at least t shares are available.
- If up to e shares are corrupted, recovery succeeds only when $m \geq t + 2e$.

The corrupted-share scenarios are especially important because they distinguish the proposed method from ordinary threshold interpolation. For example, the $(t, n, m, e) = (3, 5, 5, 1)$ scenario succeeded in all trials, whereas the $(3, 5, 4, 1)$ scenario failed in all trials because the available redundancy was one share short of the theoretical bound.

B. Failure Behavior

The invalid scenarios failed for structural rather than incidental reasons. In all runs with $m < t$, the decoder had too few constraints to identify the degree-$(t-1)$ polynomial. In all runs with $m < t + 2e$, the system lacked the redundancy needed to both identify the corrupted positions and reconstruct the correct polynomial. The implementation therefore raised decoding failure instead of returning a candidate key. This behavior is desirable because a key-recovery mechanism should fail conservatively when the mathematical recovery conditions are not satisfied.

The absence of false positives is particularly important in the corrupted-share setting. A naive reconstruction procedure that interpolates directly on all submitted points can produce a syntactically valid field element even when one or more shares are wrong. In contrast, the robust decoder in this study couples polynomial consistency checks with AES-GCM verification, so the recovery pipeline accepts only secrets that are algebraically and cryptographically consistent with the original instance.

C. Timing Behavior

The timing results show two consistent patterns. First, recovery is faster when the number of available shares is close to the minimum threshold, because the decoder solves a smaller linear system. Second, recovery time increases as either the threshold or the correctable error budget grows. This is visible when comparing the mean recovery times in Table I: the smallest successful configuration, minimum valid recovery with $(3, 5, 3, 0)$, required only 0.0736 ms on average, while the largest successful no-error configuration, $(5, 10, 10, 0)$, required 0.6451 ms.

The error-correction overhead is present but modest at the tested scale. For threshold $t = 4$, moving from a normal full-share recovery $(4, 7, 7, 0)$ to a two-error correction setting $(4, 8, 8, 2)$ increased mean recovery time from 0.2937 ms to 0.4009 ms. Likewise, for threshold $t = 5$, correcting two corrupted shares with nine available shares required 0.4787 ms on average. These values remain well below one millisecond, indicating that the algebraic overhead of robust reconstruction is practical for key-recovery workloads.

The failure cases in Fig. 2 also have a useful performance interpretation. Their measured latency is close to zero not because recovery becomes cheaper in a meaningful operational sense, but because the decoder terminates early once the share set is recognized as algebraically insufficient. This means that the additional redundancy required for robust recovery mainly affects the successful path, while invalid configurations are rejected with minimal extra computation.

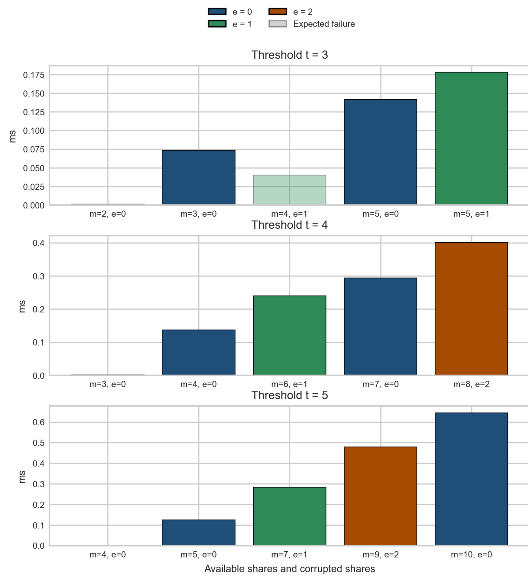

Fig. 2. Mean recovery latency across threshold settings. Darker bars denote successful configurations, while faded bars denote expected failure cases.

D. Storage Overhead

Table II summarizes storage overhead under the fixed-width share format. Because each share occupies 96 bytes, overhead grows linearly with the number of generated shares. This is a predictable trade-off: the prototype gains fault tolerance and distribution flexibility at the cost of storing more total material than the original 32-byte key.

TABLE II
STORAGE OVERHEAD IN THE PROTOTYPE

n	Bytes/Share	Total Bytes	Overhead
5	96	480	15.0×
7	96	672	21.0×
8	96	768	24.0×
10	96	960	30.0×

Although these ratios appear large compared with the original key size, the absolute storage volume remains small in practical terms. Even the largest configuration stores less than one kilobyte of share data. This is acceptable for archival key protection, password-vault backup, escrow, or multi-party administrative recovery workflows.

E. Interpretation

The results illustrate the separation between security and robustness. Increasing the threshold strengthens resistance against partial compromise but also reduces the minimum number of missing shares the system can tolerate. Adding Reed-Solomon decoding does not change that threshold policy; instead, it adds resilience against faulty inputs during reconstruction. Shamir sharing determines the access structure, whereas Reed-Solomon decoding determines the level of reconstruction fault tolerance.

Another noteworthy observation is that failure is clean. In every invalid scenario, the decoder rejected the share set rather than silently producing a usable but wrong AES key. This behavior is important because a fault-tolerant recovery system should fail closed when the algebraic constraints are unsatisfied.

The storage and timing measurements also indicate where the practical costs of the approach lie. The storage overhead is dominated by the field representation of the shares rather than by the AES key itself. The computational overhead is concentrated in the decoder, especially as the threshold and error budget grow. For the current parameter range, neither cost is prohibitive, but both would become more significant in deployments with very large thresholds or frequent reconstruction operations. This suggests that the design is best suited to backup, escrow, archival recovery, and administrative key-management settings rather than to high-frequency online use.

VII. SECURITY ANALYSIS

The confidentiality of the framework is inherited from Shamir Secret Sharing. Any adversary who obtains fewer than t valid shares learns no information about the protected AES key beyond what is already possible without the shares [1]. This is an information-theoretic guarantee of the sharing scheme and does not depend on computational hardness assumptions.

The Reed-Solomon component serves a different role. It improves availability and robustness by correcting a bounded number of corrupted shares during recovery, but it does not reduce the threshold needed to determine the secret polynomial. Therefore, it is incorrect to claim that Reed-Solomon decoding strengthens secrecy. Its benefit is operational: it allows the honest recovery process to survive a limited amount of data corruption.

A. Adversarial Capabilities

The analysis assumes that an adversary may observe, steal, delete, or modify a subset of stored shares. The adversary may also submit malformed shares during reconstruction in an attempt to induce decoder failure or to force recovery of an incorrect secret. However, the adversary is not assumed to break AES-GCM directly, invert finite-field operations, or compromise the randomness used to generate the Shamir polynomial. These assumptions match the intended application of the framework as a key-management and recovery mechanism rather than as a replacement for the underlying symmetric cipher.

The framework addresses four practical risks:

1) *Single point of failure:* no single location stores the entire AES key.
2) *Key loss due to partial unavailability:* recovery remains possible as long as the threshold or decoding bound is satisfied.
3) *Partial share compromise:* fewer than t valid shares do not reveal the secret.

4) *Limited share corruption:* bounded tampering can be corrected when enough redundancy is available.

At the same time, several limitations remain:

1) If an attacker acquires at least t valid shares, the key is recoverable by design.
2) If the reconstruction endpoint is compromised, the attacker may steal the recovered AES key after successful recovery.
3) The current prototype assumes reliable share authenticity is not provided externally. A malicious actor can still force denial of service by injecting too many invalid shares.
4) Share storage remains meaningful only if the shares are actually separated across different administrative or physical domains.

B. Mitigations and Extensions

Several extensions would strengthen the framework in deployment. Verifiable secret sharing can help participants detect incorrect shares before interpolation, while authenticated or signed share containers can make the corruption source easier to identify [6]. Operationally, share placement policies should ensure that independent shares are not stored in a single cloud account, server cluster, or administrative domain. Reconstruction should also occur only on a trusted endpoint, because the threshold mechanism protects the stored key but not a machine that is already compromised at the moment of recovery.

VIII. Conclusion

This paper presented a practical key recovery framework that combines Shamir Secret Sharing with Reed-Solomon-style Berlekamp-Welch decoding. The implementation protects an AES-256 key by distributing it into threshold shares and reconstructing it robustly in the presence of missing or corrupted inputs. A modular Python prototype was developed and evaluated on fifteen scenarios covering nominal recovery, threshold-limited failure, and corrupted-share recovery.

The experimental results matched the theoretical bounds exactly across 375 trials. Recovery succeeded in every valid scenario and failed in every invalid scenario, while mean recovery times remained below one millisecond for all tested configurations. These findings show that robust reconstruction can improve the availability of threshold key management without weakening its confidentiality guarantees.

Future work can extend the prototype with share authentication, verifiable secret sharing, compact serialization, and larger-scale performance measurements. The present results already show that coding-theoretic decoding can be integrated into threshold key recovery without altering the underlying secrecy threshold.

Acknowledgment

Praise and gratitude are only to Allah Swt., for it is through His blessings and abundant grace that the author has been able to complete this paper successfully. Special thanks are also extended to Dr. Ir. Rinaldi Munir, M.T., for his guidance and teaching across the author's studies through the sixth semester, including II4021 Cryptography, which enabled the successful completion of this paper. Additionally, heartfelt thanks are conveyed to the parents for their constant support and motivation provided to the author.

References

[1] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979, accessed: May 12, 2026. [Online]. Available: https://doi.org/10.1145/359168.359176

[2] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960, accessed: May 14, 2026. [Online]. Available: https://doi.org/10.1137/0108018

[3] S. Fedorenko, "A simple algorithm for decoding reed-solomon codes and its relation to the welch-berlekamp algorithm," 2005, accessed: May 16, 2026. [Online]. Available: https://arxiv.org/abs/cs/0501011

[4] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," National Institute of Standards and Technology, Federal Information Processing Standards Publication 197, 2023, accessed: May 15, 2026. [Online]. Available: https://csrc.nist.gov/pubs/fips/197/final

[5] M. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and GMAC," National Institute of Standards and Technology, NIST Special Publication 800-38D, 2007, accessed: May 13, 2026. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38d.pdf

[6] M. Cheraghchi, "Nearly optimal robust secret sharing," *Designs, Codes and Cryptography*, vol. 87, no. 8, pp. 1777–1796, 2019, accessed: May 12, 2026. [Online]. Available: https://doi.org/10.1007/s10623-018-0578-y

Repository

The implementation code, experimental scripts, and document source for this paper are publicly available at the following GitHub repository:

https://github.com/Nayekah/reed-solomon-secret-sharing

Statement

I hereby declare that this paper is my own work, is not a translation or reproduction of another person's paper, and is not plagiarized.

Bandung, May 18th 2026

Nayaka Ghana Subrata

NIM 13523090