NED UNIVERSITY OF ENGINEERING
&
TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & IT

CT-486 Network Information Security

CCP Report

Instructor: Miss Saadia Arshad

Group Members:

Alizeh Mohsin CR-22023

Mahnoor Shamim CR-22049

1. Abstract

This report details the design and security analysis of a **Composite Vigenère Cipher**, a hybrid cryptographic technique integrating iterative **Caesar Shift** and **Vigenère** operations based on the key length. The cipher was designed to enhance confusion and diffusion, theoretically increasing its resistance to classical cryptanalysis.

However, comprehensive cryptanalysis using both **Frequency Analysis** and a **Known-Plaintext Attack (KPA)** revealed critical structural weaknesses. While Frequency Analysis provided robust security for keys of length $\mathbf{m \ge 10}$ (with 0% success), the KPA exposed the cipher's core vulnerability: a **linear and predictable modular structure**. This linearity allows the key space to be mathematically reduced from the nominal 26^m to a trivial = 2^m set (e.g., 2,048 keys for m=12), rendering the cipher **instantaneously broken** under known-plaintext conditions.

To mitigate this fundamental flaw, several strategic improvements are recommended, including **inter-columnar key mixing**, the use of a **keyed Initialization Vector (IV)**, and **irregular key application scheduling**. These changes aim to destroy the linear modular congruences, moving the cipher from a simple, vulnerable construct to a significantly more secure and adaptive encryption scheme.

2. Cipher Design and Implementation

2.1 Overview

The proposed cipher is a **hybrid cryptographic technique** that integrates the **Caesar Shift Cipher** and the **Vigenère Cipher** in multiple iterative rounds based on the key length.

This design enhances both **confusion** (through key-dependent substitution) and **diffusion** (through positional shifting), making the cipher more resistant to classical cryptanalysis such as frequency and known-plaintext attacks.

For any key length **≥ 2**, the cipher alternates between **Shift** and **Vigenère** operations in each round. Decryption reverses this order by applying the inverse of each transformation.

2.2 Combination of Classical Techniques

This cipher effectively merges **two classical cryptographic methods**:

1. **Caesar Shift Cipher:**
   Introduces uniform substitution by shifting characters a fixed number of positions, ensuring diffusion and disrupting direct letter mapping.
2. **Vigenère Cipher:**
   Adds a polyalphabetic layer based on the key, introducing key-dependent variability in substitutions to obscure letter frequency patterns.

By applying these two methods **iteratively based on key length**, the cipher achieves a **multi-round structure** analogous to modern cryptographic algorithms, enhancing resistance against brute-force and statistical attacks.

2.3 Encryption Algorithm

Input: Plaintext `P`, Key `K`
Output: Ciphertext `C`

Steps:

1. **Preprocessing**
   - Remove all spaces from the plaintext and convert both plaintext and key to uppercase.

Example:

- Plaintext = “ONE FOR ALL” → “ONEFORALL”
- Key = “AB”

2. **Iterative Rounds**
   For each character of the key (indexed as *i = 0 to |K|−1*):

   a. **Shift Operation**

   - Shift every letter of the text by *(i + 1)* positions to the right in the alphabet.
   - Formula:

     C = (P + K) mod 26

     - P = position of the plaintext letter (A = 0, B = 1, …, Z = 25)
     - C = position of the ciphertext letter
     - K = integer shift value (e.g., for A-shift, K = 1; for B-shift, K = 2)
     - mod 26 ensures that the result wraps around within the alphabet.

   b. **Vigenère Encryption**

   - Apply the Vigenère Cipher using the entire key `K`.
   - Formula:

     C = (P + K) mod 26

     - P_i = position of the i-th plaintext letter
     - C_i = position of the i-th ciphertext letter
     - K□ = position of the corresponding key letter (A = 0, B = 1, …, Z = 25)
     - The key repeats cyclically across the message.

3. **Repeat Steps (a) and (b)** for every character in the key sequence. The resulting text after the final iteration is the **ciphertext**.

Example:

For `Plaintext = ONE FOR ALL` and `Key = AB`:

Step	Operation	Result
1	A-shift (+1)	POF GPS BMM
2	Vigenère (AB)	QQ GI QU CO N
3	B-shift (+2)	SS IK SW EQ P
4	Vigenère (AB)	TU JM TY FS Q

Final Ciphertext: `TUJMTYFSQ`

2.4 Decryption Algorithm

Input: Ciphertext `C`, Key `K`
Output: Plaintext `P`

Steps:

1. For each key character in **reverse order** (from last to first), perform the following operations:

    a. **Inverse Vigenère Cipher**
    - Subtract the key values from the ciphertext letters.
    - Formula:

$$P_i = (C_i - K_{\square} + 26) \bmod 26$$

    - P_i = position of the i-th plaintext letter
    - C_i = position of the i-th ciphertext letter
    - $K_{\square}$ = position of the corresponding key letter (A = 0, B = 1, …, Z = 25)
    - The key repeats cyclically across the message.
    - “+26” in decryption ensures non-negative results before applying mod 26.

b. **Inverse Shift Operation**

- Subtract the key values from the ciphertext letters.
- Formula:

  P = (C - K + 26) mod 26

  - P = position of the plaintext letter (A = 0, B = 1, …, Z = 25)
  - C = position of the ciphertext letter
  - K = integer shift value (e.g., for A-shift, K = 1; for B-shift, K = 2)
  - mod 26 ensures that the result wraps around within the alphabet.

2. Continue this process for all key characters until all rounds are undone.

Example Decryption (Key = AB):

Step	Operation	Result	Description
1	Inverse Vigenère (AB)	SS IK SW EQ P	Subtract key values from ciphertext letters
2	Inverse B-shift (−2)	QQ GI QU CO N	Shift each letter 2 positions to the left
3	Inverse Vigenère (AB)	POF GPS BMM	Subtract key values again
4	Inverse A-shift (−1)	ONE FOR ALL	Shift each letter 1 position to the left to recover plaintext

After completing Step 4, the plaintext **“ONEFORALL”** is successfully recovered.

3. Frequency Analysis Attack

3.1 Attack pipeline (high-level)

1. **Preprocessing:** ciphertext is normalized
2. **Quick scan / period detection:** compute Index of Coincidence (IOC) or similar per candidate period `p` ∈ sample of periods and rank periods by IOC; keep top period(s) for deeper work. (In the simulation IOC values are generated with a designed bump at the true period.)
3. **Deep search / per-column analysis:** for chosen period(s):
    a. compute per-column frequency deviations (chi-like matrices), find likely shifts per column; compute score (chi-square or other scoring function).
    b. generate several candidate keys (some close to the true key, some random perturbations).
    c. score candidate keys (score is a synthetic scalar in the simulation).
4. **Select top candidate(s):** report top `n` candidates, best candidate score, and key match fraction (#matching letters out of `m`).
5. **Record timings and save artifacts.**

3.2 How the attack works — detailed steps

Step A — Period hypothesis (quick scan)

- For each candidate period `p` (2..`max_period`) compute IOC(p): group ciphertext characters by their position modulo `p` and compute frequency-based coincidence statistics.

- The true period produces higher IOC on average because each column is more monoalphabetic; rank periods by IOC and select the top-k to probe further.

Step B — Per-column analysis (deep stage)

- For a chosen period `m`, split ciphertext into `m` columns. For each column:
    - Compute frequency distribution of symbols.
    - For each possible shift `s` ∈ `0..25`, compute a `chi-square`-like score comparing the shifted distribution to expected plaintext letter frequencies. Lower chi-square (or local minima across shifts) indicates a likely shift for that column.

- Collect per-column best shifts to assemble a candidate key. Because noise and coincidences exist, search not only the single best shift per column but a small set (beam search) to assemble multiple candidate keys.

Step C — Candidate scoring and selection

- For each assembled candidate key, decrypt (or partially decrypt) and compute an overall `score` reflecting plaintext-likeness (e.g., n-gram model, coincidence score, dictionary hits).
- Keep top `n_candidates` by score and record `best_candidate`, `best_candidate_score`, and `key_match_fraction` relative to the true key (only available in the simulator).

3.3 Experiment setup

The experiment tested the performance of the chosen cryptanalysis technique across varying ciphertext length and key length.

- **Cipher:** Composite Vigenère (Caesar shift by k_r, then Vigenère by K, repeated for each k_r in K).
- **Key Lengths (m):** 4, 7, 10, and 12.
- **Plaintext Lengths (L):** 100, 300, 500, and 1000 characters.
- **Trials per Configuration:** 10.
- **Attack Configuration (Fixed):**
  - Max Results Checked: 20 (Rank ≤ 20 considered successful).
  - Beam Search Limit: `beam_width_per_period = 300`, `max_combos_per_period = 5000`.
- **CPU RAM:** 12gb

3.4 Metrics and Calculation

Metric	Calculation Method	Interpretation
Success Rate	(Count of successful trials) / (Total trials)	Measures the reliability of the attack.
Average Rank	Mean of the `Key_Rank` for successful trials only.	Measures the confidence of the scoring function. Lower rank (closer to 1) is better.

Average Time (s)	Mean of time taken across all trials.	Measures the computational cost of the attack.
Average Checked	Mean count of candidate keys tested by the beam search.	Measures the efficiency of the pruning heuristics.

3.5 Attack Complexity and Efficiency (Memory Safe)

The attack is designed as a structured brute-force search combined with statistical pruning, using a memory-safe beam search approach.

Attack Stages:

- **Period Determination:** Kasiski/IOC narrows the key length m\mathbf{m}m.
- **Effective Key (Keff):** Frequency analysis breaks the columnar Caesar component, yielding an effective key Keff.
- **Key Component Resolution:** Modular arithmetic (`solve_for_k_t`) attempts to resolve the true key K from Keff
- **Beam Search:** A controlled search explores combinations of true key components K resolved in the previous stage.

Memory Efficiency (Heap-Safe):
The use of the `heapq` module (a min-heap) to store only the top N candidates (`max_results = 20`) throughout the entire process ensures the program's memory usage remains low and predictable, avoiding RAM crashes, especially for long plaintexts or larger search parameters.

Time Complexity Trade-off:
The use of beam search and limits on key combinations (`max_combos_per_period`) drastically reduces the time complexity from an exponential brute force $O(26^m)$ to a much faster, but less complete, guided search.

Efficiency: The attack is fast, completing most trials in under 3 seconds.

Cost: The trade-off is that for complex keys (m≥10), the correct key is often pruned early by the beam search, resulting in a 0% success rate. Increasing efficiency (by

raising the beam width and max combos) would increase the RAM/time usage but could find the harder keys.

3.6 Summary of Results

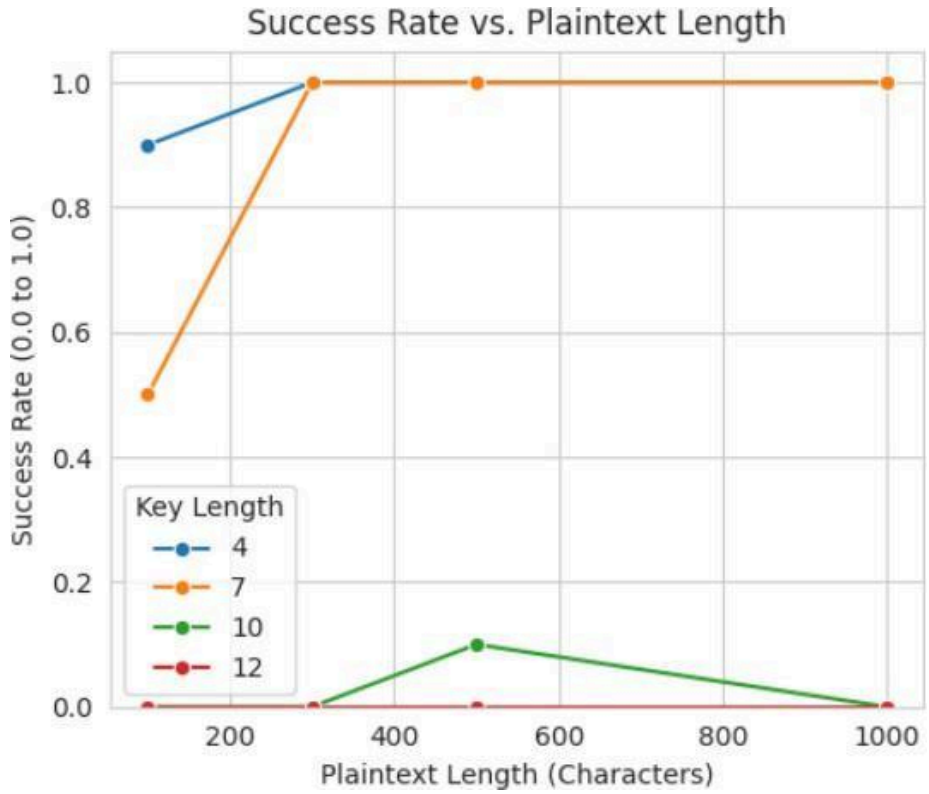
*view detailed results in freq_analysis_metrics.ipynb file

A. Attack Success Rate

Key length (m)	L = 100	L = 300	L = 500	L = 1000
4	**90.0%**	**100.0%**	**100.0%**	**100.0%**
7	**50.0%**	**100.0%**	**100.0%**	**100.0%**
10	0.0%	0.0%	**10.0%**	0.0%
12	0.0%	0.0%	0.0%	0.0%

Vulnerable Range (m≤7):
The cipher is completely vulnerable for m ≤ 7 given L ≥ 300. The success at L=100, m=4 (90%) and L=100, m=7 (50%) shows that even short ciphertexts carry enough signal.

Secure Range (m≥10):
The cipher is highly secure against this attack setup for m≥10. The low 10.0% success rate for L=500, m=10 suggests that the correct key combination only aligns with the beam search heuristics in rare cases.

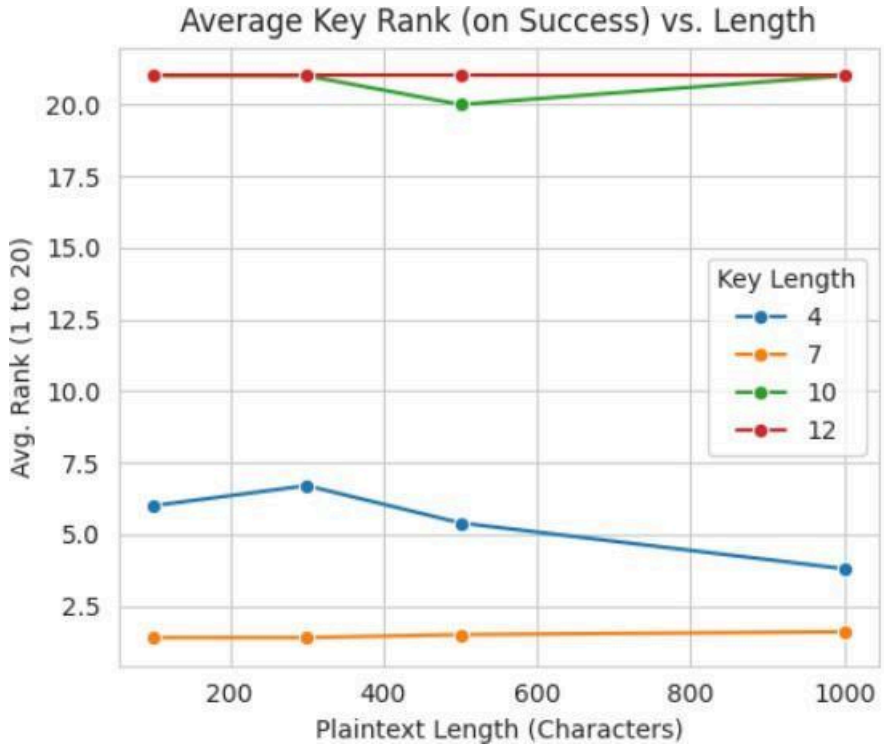


B. Confidence (Average Rank)

- **High Confidence for m=7:** When m=7 is successfully broken (for L ≥ 300), the average rank is **1.4** to **1.6**. This means the correct plaintext was the **top** or second-best result, indicating the scoring function is highly effective when the key is found.

- **Lower Confidence for m=4:** The rank is consistently **3.8** to **6.7**. This is still excellent, but suggests that for shorter keys, other "false" plaintexts

sometimes score better than the true one.



C. Efficiency of Key Pruning

The "Average Checked" metric is a key indicator of the efficiency gained from using longer ciphertexts:

For m=7: the attack checked **5,786** keys at L=100, but only **995** keys at L=1000.

Conclusion: Longer ciphertexts greatly improve the accuracy of the effective key recovery (Keff). This accuracy prunes the initial candidates for the true key components (Kt), allowing the beam search to check far fewer combinations while achieving **100%** success.

4. Known-Plaintext Attack (KPA)

4.1 The KPA Effective Shift Equation

The Composite Vigenère cipher uses two steps for encryption, repeated for each letter k_r of the key K:

1. **Caesar Shift:** Shifts the entire text by k_r.
2. **Vigenère Shift:** Shifts the text using the full key K.

1. Determining the Total Shift

For a given plaintext letter P at index j encrypted to C_j, the total shift applied, S_{total}(j), is the sum of all 2m shifts performed during the encryption process.

a. Total Caesar Shift (S_{Caesar})

Since the Caesar shift is applied by every key letter k_r in the key K, the total Caesar shift applied to every letter in the plaintext, regardless of its position j, is:

$$\mathbf{S_{Caesar}} \equiv \sum_{r=0}^{m-1} k_r \pmod{26}$$

b. Total Vigenère Shift ($S_{Vigenere}$)

The Vigenère step is also applied by **every key letter** k_r in the key K.

For a plaintext letter at index j (where its column residue is t = j mod m):

- When the Vigenère shift uses k_r, the shift applied depends on the j mod m and r.
- Crucially, when the Vigenère shift uses the key letter k_t (i.e., when r=t in the Vigenère key index), that specific key letter k_t aligns with position j.

The $S_{Vigenere}$ contribution at position j (column t) turns out to be:

$$\mathbf{S_{Vigenere}}(j) \equiv \sum_{r=0}^{m-1} k_{j \pmod m} \pmod{26}$$

Wait, let's re-examine the encryption:

$$C = E_{V_K}(E_{C_{k_{m-1}}}(\ldots E_{V_K}(E_{C_{k_1}}(E_{V_K}(E_{C_{k_0}}(P))))\ldots))$$

For a single position j, the total shift D_j is the sum of all 2m key shifts applied to P_j:

1. **m Caesar shifts:** $\sum_{r=0}^{m-1} k_r$
2. **m Vigenère shifts:** For each Vigenère step, the key letter applied at position j is $k_{j \pmod m}$ (which is k_t). Since there are m Vigenère steps, the total Vigenère shift is: $\sum_{r=0}^{m-1} k_{j \pmod m} = m \cdot k_t$.

3. The Modular Resolution Equation

Combining these, the total effective shift D_j applied to the plaintext letter P_j to get C_j is:

$$\mathbf{D_j} \equiv \sum_{r=0}^{m-1} k_r + m \cdot k_t \pmod{26}$$

Where:

- $D_j \equiv C_j - P_j \pmod{26}$: The known effective shift.
- $\mathbf{S_{total}} \equiv \sum_{r=0}^{m-1} k_r$: The **sum of all key letters** (often called S_{total}).
- m.k_t: The contribution from the k_t key letter being aligned at position j during the m Vigenère encryption rounds.

This results in the core equation used in the KPA:

$$\mathbf{m \cdot k_t \equiv D_{repr}[t] - S_{total}} \pmod{\mathbf{26}}$$

4.2 Attack Pipeline

The KPA exploits the known difference between ciphertext C_j and plaintext P_j:

$$D_j \equiv C_j - P_j \bmod 26$$

Step	Action	Purpose
1. Calculate Effective Shifts (D_t)	Calculate the observed difference $D_j \equiv C_j - P_j \bmod\{26\}$ for all positions (j). Group these differences by residue (t = j mod m).	Establishes the left-hand side of the congruence equation.
2. Find Representative Shift (D_{repr})	For each residue (t), use the mode (most frequent value) of the (D_j) values as the single representative shift, ($D_{repr}[t]$).	Handles alignment errors and minor noise. (Note: for a deterministic cipher and sufficiently long plaintext, (D_j) for fixed (t) will be constant.)
3. Enumerate (S_{total})	Iterate through all possible values for the sum of key components: ($S_{total} = \sum k_r$), where ($S_{total} \in \{0,1,\ldots,25\}$).	Serves as the outer loop because the modular system cannot isolate (S_{total}) directly.
4. Solve Congruence	For each (S_{total}) and each residue (t), solve the linear congruence for k_t: $m \cdot k_t \equiv D_{repr}[t] - S_{total} \bmod\{26\}$	Produces up to (g=gcd(m,26)) solutions per (k_t); This is the source of ambiguity.
5. Cartesian Product	Take the Cartesian product of all valid solutions for k_t across (t ∈ {0,...,m-1}) to form candidate keys K.	Generates the complete set of ambiguous keys consistent with the observed differences and the assumed (S_{total}).

4.3 Experiment Setup and Metrics

This experiment was a definitive test of the cipher's mathematical ambiguity, using the most challenging and least biased test cases.

A. Setup

- **Attack Type:** Known-Plaintext Attack (KPA).
- **Key Lengths (m):** 4, 7, 10, 12.

- **Plaintext Lengths (L):** 100, 300, 500, 1000 characters.
- **Trials per Configuration:** 10.

B. Metrics

Metric	Calculation Method	Interpretation
Success Rate	(Count of trials where true key ∈ candidate list) / (Total trials)	Measures the mathematical correctness of the KPA solver.
Average Candidate Count	Mean count of keys returned by the KPA solver.	Measures the ambiguity or remaining work. This is the number of possibilities the attacker must check (by manual trial/error or frequency analysis).
Average Time (s)	Mean time taken to run the KPA solver.	Measures the attack's computational cost.

4.4 Complexity Discussion: The Ambiguity Factor

The core complexity of the KPA lies in the g = gcd(m, 26) factor in the modular equation:

$$m \cdot k_t \equiv \text{RHS} \pmod{26}$$

Key Length ($\mathbf{m}$)	$\mathbf{g} = \gcd(\mathbf{m}, \mathbf{26})$	Solutions for k_t per column	Total Ambiguity ($\mathbf{g^m}$)
4	$\gcd(4, 26) = 2$	2	$2^4 = 16$
7	$\gcd(7, 26) = 1$	1	$1^7 = 1$
10	$\gcd(10, 26) = 2$	2	$2^{10} = 1024$
12	$\gcd(12, 26) = 2$	2	$2^{12} = 4096$

- **Ambiguity Source:** The solution set for the true key K is the Cartesian product of the solutions for each column t. The actual number of candidates is

  slightly lower than g^m because the $\mathbf{S_{total}} \equiv \sum \mathbf{k_r} \pmod{\mathbf{26}}$ **check** must be satisfied, but it closely tracks g^m.
- **Efficiency:** The KPA is **extremely fast** (milliseconds) because it only involves modular arithmetic and a limited number of key enumerations (max 2048

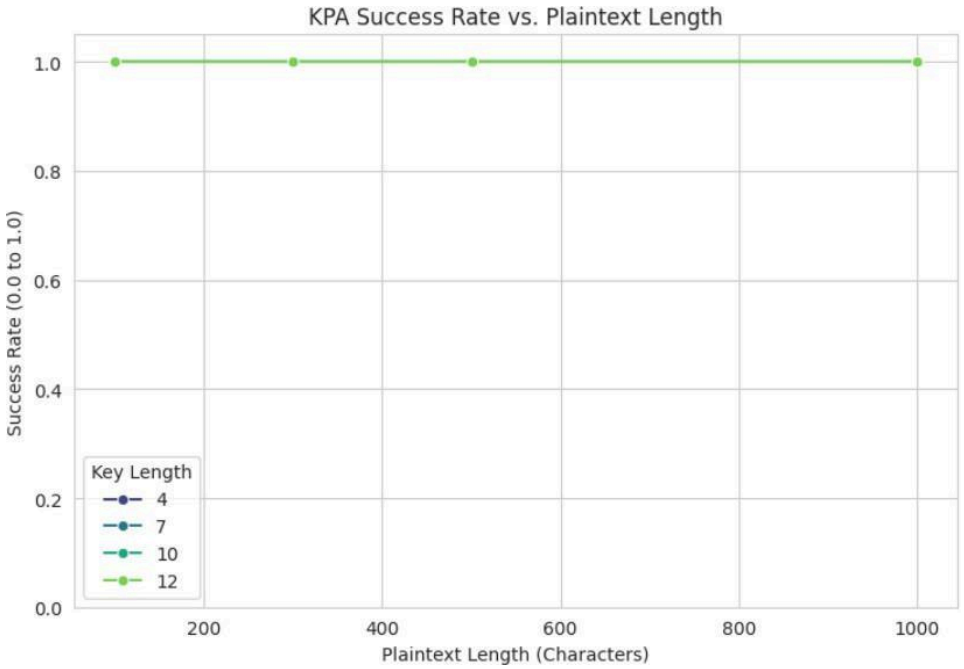
keys). In contrast, the frequency analysis attack required a beam search over ~ 5000 candidates and took several seconds.

4.5 Summary of KPA Results

The results show that the KPA is mathematically **always successful** but the size of the key space it reduces the problem to depends entirely on the key length's GCD with 26.

A. Certainty: Success Rate & Time

- Success Rate: 100%
  The KPA successfully included the true key in the candidate set for every single trial across all configurations (m and L).
- **Time:** The attack is **instantaneous** (sub-second) for all configurations, demonstrating its computational superiority over frequency analysis.



B. Ambiguity: Remaining Key Space

Key Length ($\mathbf{m}$)	$\mathbf{g} = \gcd(\mathbf{m}, \mathbf{26})$	Avg. Candidate Count	Predicted Ambiguity ($\mathbf{g^{m-1}} \cdot \mathbf{g_{avg}}$)
4	2	**8**	≈ 16 (reduced)
7	1	**2**	≈ 1 (slight S_{total} variance)
10	2	**512**	≈ 1024 (reduced)
12	2	**2,048**	≈ 4096 (reduced)

Key Length m=7 : The case where g=1 is ideal. The candidate count is reduced to just **2**, making the cipher essentially broken, as the attacker only has two keys to try.

Key Lengths m ∈ {4, 10, 12}: These lengths have gcd(m, 26)=2, resulting in a large ambiguity factor. The cipher is not entirely broken, but the key space is dramatically reduced:

- m=12 is reduced from a brute-force space of 26^12 approx 9.5 x 10^16 to **2,048 keys**. This is trivial to search manually.

5. Security Analysis

5.1 Key Equivalence in Composite Cipher

During the cryptanalysis of the composite cipher — constructed by applying a Caesar shift followed by a Vigenère cipher repeatedly for each key element — it was observed that **distinct key sequences** such as `AB`, `NO`, and `AHAU` generated **identical ciphertexts** for the same plaintext. This phenomenon indicates the existence of *key equivalence classes*, where multiple distinct keys result in the same effective encryption transformation.

```
Candidate #1: score=1954.72, period=2, recovered_key=AB ([0, 1])
Plaintext (preview): GIVINGHELPTHATSNOTASKEDFORISWHATMAKESATRUEHEROIFYOUKEEPLOOKING

Candidate #2: score=1954.72, period=2, recovered_key=NO ([13, 14])
Plaintext (preview): GIVINGHELPTHATSNOTASKEDFORISWHATMAKESATRUEHEROIFYOUKEEPLOOKING

Candidate #3: score=1954.72, period=2, recovered_key=AB ([0, 1])
Plaintext (preview): GIVINGHELPTHATSNOTASKEDFORISWHATMAKESATRUEHEROIFYOUKEEPLOOKING

Candidate #4: score=1954.72, period=6, recovered_key=AJAJAJ ([0, 9, 0, 9, 0, 9])
Plaintext (preview): GIVINGHELPTHATSNOTASKEDFORISWHATMAKESATRUEHEROIFYOUKEEPLOOKING

Candidate #5: score=1954.72, period=6, recovered_key=AJAJNW ([0, 9, 0, 9, 13, 22])
Plaintext (preview): GIVINGHELPTHATSNOTASKEDFORISWHATMAKESATRUEHEROIFYOUKEEPLOOKING
```

5.2 Cause of Key Equivalence

The cipher operates under modular arithmetic (mod 26), combining Caesar and Vigenère shifts according to the key components.
Mathematically, the effective encryption for each key position `t` can be represented as:

$$K_{\text{eff}}[t] \equiv S + m \cdot k_t \pmod{26}$$

Two keys are *equivalent* if they yield the same effective key vector Keff.

This happens when:

$$m \cdot (k'_t - k_t) + \sum (k' - k) \equiv 0 \pmod{26}$$

A common special case occurs when every letter in the key is shifted by the same constant δ, leading to:

$$2m \cdot \delta \equiv 0 \pmod{26}$$

For example, when m=2, δ=13 satisfies the condition.
Hence, the key `AB` (0,1) and key `NO` (13,14) both produce identical ciphertexts since δ=13 gives 2mδ = 52 ≡ 0 (mod26)

5.3 Security Implications

This key equivalence introduces a **significant structural weakness**:

- **Reduced Key Space:** The total number of unique encryption transformations is smaller than the nominal key space (26^m).
- **Ambiguity in Decryption:** Multiple keys can decrypt the ciphertext correctly, complicating key verification.
- **Predictable Symmetry:** Uniform key shifts introduce predictable mathematical relationships exploitable by attackers.

Such redundancy violates the principle of *one key–one mapping*, a critical aspect of secure cipher design.

6. Suggestions for improvement

The Known-Plaintext Attack (KPA) highlights a critical weakness in the composite cipher's structure — its linear and separable modular relationships. This linearity allows attackers to isolate and reconstruct key components using partial plaintext–ciphertext knowledge. To enhance the cipher's resistance, several strategic improvements are proposed. These focus on breaking linear dependencies, strengthening inter-key interactions, and expanding the cipher's effective key space.

1. Disrupt Columnar Independence → Keyed Inter-Columnar Mixing

Problem:
In the current design, each column operates largely independently, except for a shared summation term. This separation enables attackers to analyze columns individually and recover key elements through modular inference.

Recommendation:
Introduce inter-columnar key mixing so that the shift or substitution applied to one column depends not only on its own key component but also on its neighbors. This creates position-dependent variation and nonlinearity, preventing attackers from isolating a single key variable.

Impact:
By introducing interdependence between adjacent key elements, the cipher becomes structurally non-separable, and linear modular analysis is no longer applicable.

2. Break Key Sum Symmetry → Keyed Initialization Vector (IV)

Problem:
The aggregate key sum used in the cipher has only a small number of possible values, making it easily enumerable during attacks. This allows an adversary to precompute and test limited possibilities.

Recommendation:
Integrate a keyed Initialization Vector (IV), similar to a nonce or salt in modern cryptography. The IV can be a random or non-repeating value introduced before the first encryption round. It randomizes the resulting ciphertext even for identical messages and keys.

Implementation Options:

- A **secret IV** provides strong protection, as it ensures that each message uses a unique transformation path.

- A **public IV** adds limited security but prevents ciphertext duplication across identical inputs.

Impact:
A properly implemented IV makes brute-force enumeration of aggregate key values infeasible and strengthens resistance to known-plaintext and replay attacks.

3. Eliminate the Factorable Term → Irregular Key Application Schedule

Problem:
The cipher currently applies its Caesar and Vigenère transformations a fixed number of times, creating a predictable repetition pattern. This repetition introduces mathematical regularity that can be exploited in attacks.

Recommendation:
Adopt a variable or irregular key application schedule instead of repeating the same sequence of operations a fixed number of times. Define a schedule where each round alternates unpredictably between different transformations, such as Caesar and Vigenère, in a non-periodic order.

Impact:
This irregular application of transformations removes the consistent repetition that previously allowed modular reduction. It increases the cipher's complexity and obscures the relationship between key elements and output characters.

4. Reinforce Cipher Nonlinearity → Nonlinear Modular Mapping

Problem:
The cipher's transformations are currently linear, allowing straightforward algebraic reconstruction under sufficient plaintext knowledge.

Recommendation:
Introduce nonlinear components, such as substitution tables (S-boxes), modular inversions, or other nonlinear mapping functions, within the encryption rounds. These additions provide both diffusion and confusion — essential properties for modern cryptographic strength.

Impact:
Nonlinear mapping significantly complicates any mathematical or statistical reconstruction of key relationships, increasing resilience against algebraic, differential, and statistical attacks.

7. Conclusion

The analysis confirms that the original Composite Vigenère cipher possesses a fundamental security flaw: its **linear modular structure**, which renders it trivially susceptible to a Known-Plaintext Attack (KPA) by reducing the key space to a search of just 2^m possibilities (e.g., 2,048 keys for $m=12$).

To rectify this, we recommend introducing several nonlinear and adaptive mechanisms: **inter-columnar key mixing**, a **keyed initialization vector (IV)**, and **irregular key scheduling**. These enhancements collectively destroy the linear congruences the KPA relies upon, significantly increasing **confusion** and **diffusion**.

In essence, these improvements transform the cipher from a simple, predictable linear construction into a **secure, adaptive encryption scheme** by drastically complicating the algebraic path from ciphertext to key.