COMP3028 Computer Security — Coursework 1

Report: Task 1 (Frequency Analysis), Task 4 (Padding), Task 5 (Error Propagation)

David Leong Jin Tao
Student ID: 20620891
`https://github.com/davidjt01/comp3028-coursework1`

February 26, 2026

Contents

Part I

Task 1: Frequency Analysis

1 Introduction

1.1 Task Statement

This report addresses Task 1 of COMP3028 Coursework 1: to decipher the substitution ciphertext provided on Moodle and to recover the encryption key. The ciphertext was produced by a *monoalphabetic substitution cipher*, in which each plaintext letter is replaced by a unique ciphertext letter according to a fixed permutation of the 26-letter alphabet.

1.2 Scope and Structure

The report presents the approach adopted, the implementation, and the results. It is organised as follows: (i) the use of monogram, bigram, and trigram frequency data to construct and score candidate keys; (ii) the methodology, including the initial key from frequency ranking and an optional crib, hill climbing with multiple restarts, exhaustive pair-swap refinement, and limited manual correction; (iii) the implementation, with relevant code and justification; (iv) the recovered encryption key and decrypted plaintext, with supporting program output; (v) observations and a comparison with the SEED Lab exercise; and (vi) conclusions.

2 Literature Review

Established methods for breaking monoalphabetic substitution ciphers rely on n-gram frequency analysis: monograms (single letters), bigrams, trigrams, and optionally quadgrams [1, 2, 3].

2.1 Role of n-gram statistics

In English, letter frequencies follow a characteristic distribution (e.g. E, T, A, O, I, N, S, H, R, . . .) [1]. Mapping the most frequent cipher letter to E, the second to T, and so on yields a first approximation of the key. Bigrams (e.g. TH, HE, IN, ER) and trigrams (e.g. THE, AND, ING) exhibit more skewed distributions; correct plaintext contains many high-probability n-grams, whereas incorrect keys produce rare n-grams [2]. Quadgrams further improve discrimination at the cost of additional data and computation [3]. Reference statistics are typically drawn from corpora such as Norvig's letter n-gram counts [4].

2.2 Fitness (scoring) functions

A fitness function assigns a *higher* score to text that is more English-like. Probabilities (or log-probabilities) for 1-grams, 2-grams, and 3-grams are usually precomputed from a reference corpus. For a candidate decryption, each n-gram is looked up; unknown or zero-probability n-grams are assigned a small floor probability to avoid log(0). The score is typically the **sum of log-probabilities** (log-likelihood) over monograms, bigrams, and trigrams, optionally combined as a weighted sum [2, 5]. The objective is to find the key that *maximises* this fitness when applied to the ciphertext.

2.3 Search algorithms

The key space has size 26!, so exhaustive search is infeasible [1]. **Hill climbing** starts from an initial key (e.g. from frequency ranking or random) and repeatedly moves to a *neighbour* key that improves fitness; a natural neighbourhood is **swapping two letters** in the key ($\binom{26}{2} = 325$ neighbours). The process terminates at a local maximum [2]. Because hill climbing may terminate at a local optimum, **multiple restarts** from different initial keys are used, and the best key over all runs is retained [3, 2]. Some variants use targeted neighbours (e.g. swapping letters to correct the least common bigrams in the current decryption) or simulated annealing to accept worse keys with decreasing probability so as to escape local optima.

2.4 Effectiveness and limitations

The approach is effective for monoalphabetic substitution given sufficiently long ciphertext; the *unicity distance* is approximately 28 characters [3]. Short or statistically atypical plaintext may resist recovery or remain ambiguous. Success depends on the reference corpus matching the plaintext language.

Building on these ideas, the following section describes the methodology adopted for the present task.

3 Methodology

The methodology comprised the following steps.

1. **Input.** The ciphertext was read from `data/task1/article_encrypted.txt` (1530 characters in total). Non-letter characters (e.g. `[]` and newlines) were preserved unchanged; only letters a–z were subject to the substitution mapping.

2. **Reference data.** Letter (monogram), bigram, and trigram frequencies were loaded from `data/frequencies/*.json`. These JSON files can be produced by running `scripts/fetch_wikipedia_frequencies.py`, which fetches the corresponding frequency tables from Wikipedia (Letter frequency, Bigram, Trigram) and saves them in the format expected by the decrypt script. Letter frequencies defined the plaintext letter order (E, T, A, O, . . .); bigram and trigram data were converted to probabilities for the fitness function. Unknown n-grams were assigned small default probabilities (10^{-5}, 10^{-8}, 10^{-10} for unigrams, bigrams, and trigrams respectively) so as to avoid $-\infty$ in the log-sum.

3. **Initial key.** Ciphertext letter counts were computed and ranked by frequency. One initial key was constructed by mapping the most frequent cipher letter to E, the second to T, and so on. An alternative initial key was constructed using a crib when the most frequent cipher trigram (e.g. `meh`) had count at least 3: that trigram was assumed to correspond to `the`, and the remaining key positions were filled by frequency rank.

4. **Fitness function.** For each candidate key, the ciphertext was decrypted and the letters-only string was scored. Fitness was defined as a weighted sum of log-probabilities of

unigrams, bigrams, and trigrams (weights 0.2, 0.3, 0.5). Higher fitness corresponds to more English-like text.

5. **Hill climbing.** From each initial key, hill climbing was applied: two randomly chosen positions in the key were swapped, the ciphertext was decrypted, and fitness was recomputed; the swap was retained only if fitness improved. The process ran for up to 10 000 iterations and terminated after 1500 iterations without improvement. Five restarts were run, alternating between the frequency-based and crib-based initial keys; the key with the highest fitness over all runs was retained.
6. **Exhaustive refinement.** The best key was refined by enumerating all 325 pair swaps; any swap that improved fitness was retained, and the process was repeated until no further improvement was possible.
7. **Final key.** A small set of manual letter swaps was applied to correct remaining confusions (e.g. b/w, k/v), yielding a fully coherent decryption. The final key was written in the required format and used to produce the decrypted plaintext file.

3.1 Design choices

- **Trigrams and bigrams.** Trigrams were used because they discriminate strongly between correct and incorrect decryptions; the implementation also used bigrams and monograms in a single weighted fitness to stabilise the search.
- **Multiple restarts.** Hill climbing may converge to a local maximum; multiple restarts from different seeds and from both frequency-based and crib-based keys increase the probability of recovering the correct key.
- **Unknown n-grams.** Assigning a small floor probability avoids log(0) and ensures that the fitness is defined for any decryption; rare or missing n-grams then contribute a fixed negative penalty.

4 Implementation and code

The methodology above was implemented in a single Python script, `scripts/decrypt_task1.py`. The frequency JSON files used by the script are produced by a separate script, `scripts/fetch_wikipedia_frequencies.py`, which obtains letter, bigram, and trigram data from Wikipedia and saves them as JSON. The following subsections describe first how that reference data is obtained and saved, then how the decrypt script uses it: loading letter frequency data; decryption and key application; letter-only and n-gram extraction; cipher letter counts and initial key construction; unigram probabilities and safe logarithm; the fitness function; hill climbing; multiple restarts and best-key selection; exhaustive pair-swap refinement; manual swap application; and the output format.

4.1 Obtaining reference data from Wikipedia

(`scripts/fetch_wikipedia_frequencies.py`, `fetch_page` lines 29–33, `main` lines 122–147.) The letter, bigram, and trigram frequency data are fetched from the following Wikipedia pages and saved as JSON in `data/frequencies/`:

- **Letter frequency:** `https://en.wikipedia.org/wiki/Letter_frequency` — the main English letter frequency table (Letter, Relative frequency in texts, in dictionaries) is parsed from a `wikitable`; each row yields `letter`, `texts_percent`, and optionally `dictionaries_percent`.
- **Bigram:** `https://en.wikipedia.org/wiki/Bigram` — bigram percentages are taken from a `<pre>` block on the page (e.g. `th 3.56%`, `he 3.07%`); each pair is stored as `bigram` and `frequency_percent`.
- **Trigram:** `https://en.wikipedia.org/wiki/Trigram` — the trigram frequency table (Rank, Trigram, Frequency) is parsed from an HTML table; each row yields `trigram` and `frequency_percent`.

The script uses `requests` to fetch each page (with a User-Agent header) and `BeautifulSoup` to parse the HTML. Parsed data are written with `json.dump` to `data/frequencies/letter_frequency.json`, `data/frequencies/bigram_frequency.json`, and `data/frequencies/trigram_frequency.json`. The decrypt script then reads these files and expects the field names above (`texts_percent` for letters, `frequency_percent` for bigrams and trigrams).

Listing 1: Fetching a page and saving letter frequency as JSON.

```
def fetch_page(url: str) -> str:
    resp = requests.get(url, headers={"User-Agent": USER_AGENT}, timeout=30)
    resp.raise_for_status()
    return resp.text

# In main(): for each of letter, bigram, trigram:
#   html = fetch_page(URL_...)
#   data = parse_letter_frequency(html)  # or parse_bigram_*, parse_trigram_*
#   with open(OUTPUT_DIR / "letter_frequency.json", "w", encoding="utf-8") as f:
#       json.dump(data, f, indent=2)
```

4.2 Loading letter frequency data

(`scripts/decrypt_task1.py`, `load_letter_frequencies`, lines 61–87.) The script reads `letter_frequency.json`, parses each letter and its `texts_percent` value, and sorts by frequency descending to obtain the plaintext letter order (E, T, A, O, . . .). Invalid percentage values are caught and set to 0. Bigram and trigram data are loaded similarly from their JSON files (using `frequency_percent`) and converted to probability dictionaries for the fitness function.

Listing 2: Loading letter frequency and ordering by frequency.

```
with open(LETTER_FREQ_PATH, encoding="utf-8") as f:
    data = json.load(f)
out = []
for item in data:
    letter = item["letter"].upper()
    raw = item.get("texts_percent", "0%").replace("%", "").strip()
    try:
        pct = float(raw)
    except ValueError:
        pct = 0.0
    out.append((letter.lower(), pct))
out.sort(key=lambda x: -x[1])   # E, T, A, O, ...
```

4.3 Decryption and key application

(`scripts/decrypt_task1.py`, `decrypt_text`, lines 167–171.) The key is represented as a 26-character string where `key[i]` is the cipher letter for plain letter `ALPHABET[i]` (i.e. encryption: plain `a` → `key[0]`). Decryption uses the inverse mapping: for each cipher letter `c`, find the index `i` such that `key[i]==c`, then the plain letter is `ALPHABET[i]`.

Listing 3: Decryption (inverse substitution).

```
def decrypt_text(ciphertext: str, key: str) -> str:
    inv = {key[i]: ALPHABET[i] for i in range(26)}
    return "".join(inv.get(c.lower(), c) if c.isalpha() else c for c in
        ciphertext)
```

Every candidate key is evaluated by decrypting the ciphertext and scoring the result; this mapping is therefore central to the key-recovery process.

4.4 Letters-only and n-gram extraction

(`scripts/decrypt_task1.py`, lines 151–159.) The fitness function scores only letters; non-letters (e.g. `[]`, newlines) are stripped before counting. Overlapping n-grams are extracted with a sliding window so that every bigram and trigram in the decrypted text contributes to the score.

Listing 4: Letter-only text and n-gram extraction.

```
def letters_only(text: str) -> str:
    return "".join(c.lower() for c in text if c.isalpha())

def extract_ngrams(text: str, n: int) -> list[str]:
    t = letters_only(text)
    return [t[i : i + n] for i in range(len(t) - n + 1)] if len(t) >= n
        else []
```

Rationale. Cipher letter frequency ranking and the crib (most common cipher trigram) both operate on letter-only input; the fitness function uses `extract_ngrams` to obtain bigrams and trigrams from the candidate decryption. Omitting this step would allow non-letter characters to distort the statistics.

4.5 Cipher letter counts

(`scripts/decrypt_task1.py`, `cipher_letter_counts`, lines 182–188.) The ciphertext (letters only) is counted per letter and sorted by count descending. This ranking is later matched to English letter frequency order to build the initial key.

Listing 5: Rank cipher letters by frequency.

```
def cipher_letter_counts(letter_only: str) -> list[tuple[str, int]]:
    cnt = Counter(letter_only)
    return sorted(
        [(c, cnt[c]) for c in ALPHABET if cnt[c] > 0],
        key=lambda x: -x[1],
    )
```

Rationale. This function produces the cipher-side frequency ranking that `build_initial_key` maps to the plaintext letter order (E, T, A, . . .), under the assumption that the most frequent cipher letter corresponds to E, the second to T, and so on.

4.6 Building the initial key from letter frequency

(`scripts/decrypt_task1.py`, `build_initial_key`, lines 191–223.) Cipher letters are ranked by count; plain letters are ordered by reference frequency (E, T, A, . . .). The most frequent cipher letter is mapped to E, the second to T, and so on.

Listing 6: Initial key from frequency ranking (excerpt).

```
plain_ranked = [x[0] for x in letter_freq_order]
cipher_letters_ranked = [x[0] for x in cipher_ranked]
# ... build key_list so key_list[ALPHABET.index(plain)] = cipher
return "".join(key_list)
```

This provides a principled starting point for hill climbing, avoiding the need to start from an arbitrary permutation.

4.7 Crib-based initial key

(`scripts/decrypt_task1.py`, `build_initial_key_with_crib`, lines 226–270.) When the most frequent cipher trigram (e.g. `meh`) is assumed to be `the`, the three letter positions are fixed in the key; the remaining 23 positions are filled by matching plain letter frequency order to the remaining cipher letters by rank.

Listing 7: Crib: fix three positions, fill rest by frequency.

```
# Fix crib: e.g. cipher "meh" -> plain "the"
for i in range(3):
    plain = plain_trigram[i]
    cipher = cipher_trigram[i]
    idx = ALPHABET.index(plain)
    key_list[idx] = cipher
# Fill remaining positions from plain_ranked and cipher_letters_ranked
# (skip already fixed), then any gaps from unused alphabet.
```

Rationale. The crib supplies an alternative initial key that typically attains a better local maximum than the purely frequency-based key; the main loop executes hill climbing from both and retains the key with the highest fitness.

4.8 Unigram probabilities and safe log

(`scripts/decrypt_task1.py`, `safe_log`, lines 276–278; `unigram_probs_from_letter_freq`, lines 431–435.) Reference letter frequencies are normalised to probabilities (sum to 1) for the fitness. To avoid log(0) when an n-gram is missing from the reference, a floor value is used.

Listing 8: Unigram probabilities from reference; safe logarithm.

```python
def safe_log(p: float, floor: float = 1e-10) -> float:
    return math.log(max(p, floor))

def unigram_probs_from_letter_freq(letter_freq) -> dict[str, float]:
    total = sum(p for _, p in letter_freq)
    if total <= 0:
        total = 1.0
    return {letter: p / total for letter, p in letter_freq}
```

Rationale. The fitness function uses these unigram probabilities and `safe_log` for every letter and n-gram; without the floor, unknown n-grams would yield $-\infty$ and invalidate the comparison between candidate keys.

4.9 Fitness function

(`scripts/decrypt_task1.py`, `fitness`, lines 281–305.) The fitness is a weighted sum of log-probabilities for unigrams, bigrams, and trigrams in the letter-only decrypted text. Missing n-grams use default probabilities (and `safe_log` avoids $-\infty$).

Listing 9: Fitness (n-gram log-probability sum).

```python
def fitness(decrypted_letters, unigram_probs, bigram_probs, trigram_probs,
            weights=(0.2, 0.3, 0.5)) -> float:
    default_uni = 1e-5
    default_bi = 1e-8
    default_tri = 1e-10
    score = 0.0
    for c in decrypted_letters:
        score += weights[0] * safe_log(unigram_probs.get(c, default_uni))
    for bg in extract_ngrams(decrypted_letters, 2):
        score += weights[1] * safe_log(bigram_probs.get(bg, default_bi))
    for tg in extract_ngrams(decrypted_letters, 3):
        score += weights[2] * safe_log(trigram_probs.get(tg, default_tri))
    return score
```

Maximising this score is the objective of the search and is the criterion by which candidate keys are compared.

4.10 Hill climbing

(`scripts/decrypt_task1.py`, `hill_climb`, lines 311–367.) The algorithm repeatedly swaps two random positions in the key; if the new key yields higher fitness, it is retained; otherwise the swap is reverted.

Listing 10: Hill climbing with random pair swaps.

```python
i, j = random.randint(0, 25), random.randint(0, 25)
if i == j: continue
key_list[i], key_list[j] = key_list[j], key_list[i]
new_key = "".join(key_list)
```

```
new_score = fitness(letters_only(decrypt_text(cipher_letters, new_key)),
    ...)
if new_score > current_score:
    current_key = new_key
    current_score = new_score
    no_improve = 0
else:
    key_list[i], key_list[j] = key_list[j], key_list[i]
    no_improve += 1
```

This refines the key by using n-gram fitness to correct local errors that arise from monogram-only frequency mapping.

4.11 Multiple restarts and best-key selection

(`scripts/decrypt_task1.py`, `main`, lines 518–547.) Hill climbing is executed five times from different starting keys (alternating frequency-based and crib-based); each run uses a fixed random seed for reproducibility. The key with the highest fitness over all runs is retained for exhaustive refinement.

Listing 11: Multiple restarts; keep best key by fitness.

```
candidates = [(initial_key, "frequency-based")]
if initial_key_crib is not None:
    candidates.append((initial_key_crib, "crib meh->the"))
best_key, best_score = None, -float("inf")
for run in range(num_restarts):
    start_key, _ = candidates[run % len(candidates)]
    key = hill_climb(ciphertext_raw, start_key, ...)
    score = fitness(letters_only(decrypt_text(ciphertext_raw, key)),
        ...)
    if score > best_score:
        best_score, best_key = score, key
```

Rationale. A single hill-climb may terminate at a local maximum; multiple restarts from both frequency-based and crib-based initial keys, with retention of the best result, increase the probability of recovering the correct substitution.

4.12 Exhaustive pair-swap refinement

(`scripts/decrypt_task1.py`, `exhaustive_swap_refinement`, lines 372–425.) After hill climbing, every pair of indices (i, j) with $i < j$ is tried in order; the first swap that improves fitness is kept, then the full enumeration is repeated. This continues until no single swap improves fitness, reaching a local maximum over all 325 possible single swaps.

Listing 12: Exhaustive refinement: try all pair swaps.

```
key_list = list(current_key)
for i in range(26):
    for j in range(i + 1, 26):
        key_list[i], key_list[j] = key_list[j], key_list[i]
        new_key = "".join(key_list)
        new_score = fitness(letters_only(decrypt_text(ciphertext,
            new_key)), ...)
        if new_score > current_score:
                current_key = new_key
                current_score = new_score
                improved = True
                break
        else:
            key_list[i], key_list[j] = key_list[j], key_list[i]
```

Rationale. Hill climbing uses random swaps and may halt before testing every pair; exhaustive refinement guarantees that no improving single swap is overlooked, thereby correcting remaining letter confusions (e.g. b/w, p/k) that the stochastic search may have missed.

4.13 Manual swap application

(`scripts/decrypt_task1.py`, `apply_swaps`, lines 440–445.) After exhaustive refinement, a fixed list of index pairs is applied to correct remaining letter confusions (e.g. b↔w, k↔v) that the n-gram fitness did not resolve. Each pair (i, j) swaps `key[i]` and `key[j]`; the resulting key is used for the final decryption.

The swaps were applied in the following order (plain-letter indices: a=0, b=1, ..., z=25), with the reason for each:

1. $(1, 22)$: b↔w. Corrects "world" vs. "was" (e.g. "end of world war i").
2. $(10, 21)$: k↔v. Corrects "marketing" vs. "governments" (e.g. "military and government services").
3. $(1, 15)$: b↔p. Corrects "product" vs. "described" (e.g. "finished product", "described in detail").
4. $(9, 23)$: j↔x. Corrects "adjacently" (e.g. "used ... adjacently").
5. $(16, 23)$: q↔x. Corrects "complex" (e.g. "the most complex").
6. $(16, 25)$: q↔z. Corrects "blitzkrieg" and "emphasized".

Listing 13: Apply a list of index swaps to the key.

```
def apply_swaps(key: str, swaps: list[tuple[int, int]]) -> str:
    key_list = list(key)
    for i, j in swaps:
        key_list[i], key_list[j] = key_list[j], key_list[i]
    return "".join(key_list)
```

4.14 Output format

(`scripts/decrypt_task1.py`, `format_key_report`, lines 448–456.) The key is written in the coursework format: one line of plaintext letters, one line of ciphertext letters (same order).

Listing 14: Key output format.

```
def format_key_report(key: str) -> str:
    plain_line = "Plaintext  " + " ".join(ALPHABET)
    cipher_line = "Ciphertext " + " ".join(key)
    return plain_line + "\n" + cipher_line
```

This produces the key in the format required for the report.

5 Results

Application of the methodology and implementation described in Sections 3 and 4 to the provided ciphertext yielded the results below. This section presents the recovered encryption key, evidence of correct decryption (in both raw and spaced form), and quantitative summary statistics. Section 6 reproduces the full execution log.

5.1 Recovered encryption key

The recovered encryption key maps each plaintext letter to a ciphertext letter. Table 1 gives the mapping in the format required by the coursework (plaintext a–z in order, ciphertext letters in the same order).

Plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Ciphertext	g	t	q	d	h	b	j	e	i	v	z	r	c	w	s	p	f	x	u	m	k	o	l	y	n	a

Table 1: Recovered substitution key: plaintext letter → ciphertext letter (e.g. a→g, b→t, z→a).

5.2 Evidence of correct decryption

The decrypted output was written to `data/task1/article_decrypted.txt`. Because the ciphertext contained no spaces (see Section 7), the decryption is a continuous letter stream. A representative excerpt (first 500 characters) from that file is reproduced verbatim below:

```
theenigmamachinewasinventedbythegermanengineerarthurscherbiusatthe
endofworldwari[]thiswasnotknownuntilwhenapaperbykarlde
leeuwwasfoundthatdescribedindetailarthurscherbiuschanges[]the
germanfirmscherbiusrittercofoundedbyarthurscherbiuspatentedideas
foraciphermachineinandbeganmarketingthefinishedproductunder
thebrandnameenigmaininitiallytargetedatcommercialmarkets[]
earlymodelswereusedcommerciallyfromtheearlysandadoptedby
militaryandgovernmentservicesofseveralcountriesmostnotablynazi
germanybeforeandduringworldwarii

severaldifferentenigmamodelswereproducedbutthegermanmilitarymodels
havingaplugboardwerethemostcomplexjapaneseanditalianmodelswere
```

The content is recognisably about the Enigma machine (Arthur Scherbius, World War I and II, commercial and military use). The `[]` symbols are preserved from the ciphertext as non-letter characters that were not substituted.

To demonstrate that this letter stream corresponds to readable English, the same plaintext with word boundaries restored (from `data/task1/article_decrypted_spaced.txt`) is given below. The decryption is unchanged; spacing has been added for readability only.

```
the enigma machine was invented by the german engineer arthur scherbius at the
end of world war i [] this was not known until when a paper by karl de
leeuw was found that described in detail arthur scherbius changes [] the
german firm scherbius ritter co founded by arthur scherbius patented ideas
```

```
for a cipher machine in and began marketing the finished product under
the brand name enigma in initially targeted at commercial markets []
early models were used commercially from the early s and adopted by
military and government services of several countries most notably nazi
germany before and during world war ii

several different enigma models were produced but the german military models
having a plugboard were the most complex japanese and italian models were
also in use with its adoption in slightly modified form by the german navy
in and the german army and air force soon after the name enigma became
widely known in military circles pre war german military planning emphasized
fast mobile forces and tactics later known as blitzkrieg which depend on
radio communication for command and coordination since adversaries would
likely intercept radio signals messages would have to be protected with
secure encoding compact and easily portable the enigma machine filled that
need
```

5.3 Quantitative results

- Ciphertext length: 1530 characters.
- Cipher letter counts (top 5): h=188, g=142, w=120, i=118, x=109.
- Most common cipher trigram: `meh` (count 21), used as crib for `the`.
- Restarts: 5 (alternating frequency-based and crib-based initial keys).
- Final fitness (best run): Restarts 2 and 4 (crib-based) achieved fitness values of approximately −22974 to −22977; frequency-based restarts achieved approximately −23921 to −24100.
- Exhaustive pair-swap refinement was applied after hill climbing; manual swaps were then applied to achieve full coherence.

6 Program output and execution log

The key and decrypted text presented in Section 5 were produced by executing `pythonscripts/decrypt_task1.py`. The output below confirms the recovered key, manual swaps, and decrypted plaintext. To satisfy the requirement for complete steps with justification, the full terminal output is reproduced below as plain text (no screenshots). The log is divided into two parts: the heuristic phase (frequency analysis, initial key construction, hill climbing, and exhaustive refinement), and the manual phase (letter-swap corrections and final key).

6.1 Heuristic phase: from loading data to exhaustive refinement

The execution log from script start through exhaustive pair-swap refinement is given below. Each phase includes the STEP and JUSTIFICATION output by the script.

```
COMP3028 Task 1: Substitution cipher decryption using frequency analysis
Using provided letter, bigram and trigram data with full process logging.

```

```
============================================================
  Loading letter frequency data
============================================================
  Loaded 26 letters; order by frequency (highest first): e t a o i n s h r d ...
  STEP: Use letter frequency order to guess substitution.
  JUSTIFICATION: In English, E is most common (~12.7%), then T (~9.1%), A, O, I,
  N, etc. We will map the most frequent cipher letter to E, the second to T,
  and so on.

============================================================
  Loading bigram frequency data
============================================================
  Loaded 42 bigrams; e.g. th=0.0356, he=0.0307
  STEP: Use bigram frequencies to score and refine the key.
  JUSTIFICATION: Common English bigrams (e.g. th, he, in, er) should appear often
  in correct plaintext; wrong keys produce rare bigrams. We score decrypted text
  by sum of log(bigram_prob).

============================================================
  Loading trigram frequency data
============================================================
  Loaded 16 trigrams; e.g. the=0.0181, and=0.0073
  STEP: Use trigram frequencies to score and refine the key.
  JUSTIFICATION: Trigrams like 'the', 'and', 'ing' are very common in English;
  incorrect decryptions contain rare trigrams and get a lower score.

============================================================
  Loading ciphertext
============================================================
  Loaded 1530 characters from article_encrypted.txt
  STEP: We preserve the full ciphertext including non-letters (e.g. []).
  JUSTIFICATION: Non-letters are left unchanged when applying the substitution;
  only a-z are decrypted.

============================================================
  Analysing ciphertext letter frequencies
============================================================
  Cipher letter counts (top 10): h=188, g=142, w=120, i=118, x=109, m=107,
  s=89, u=86, d=70, r=63
  STEP: Count how often each cipher letter appears.
  JUSTIFICATION: This ranking is matched to English letter frequency ranking to
  form the initial key.

============================================================
  Building initial key from letter frequency (monogram)
============================================================
  Plain (by freq): e t a o i n s h r d l c ...
  Cipher (by freq): h g w i x m s u d r c q ...
  STEP: Map cipher letters to plain letters by matching frequency rank.
  JUSTIFICATION: The most frequent cipher letter is assumed to be E, the second
  T, etc. This gives a first approximation of the substitution key.

  Most common cipher trigram: 'meh' (count=21)
  STEP: Also build an alternative initial key using crib: 'meh' -> 'the'.
  JUSTIFICATION: The most frequent trigram in English is 'the'; using it to
  seed the key often improves the result.

============================================================
  Refining key with hill climbing (bigram and trigram)
============================================================
  STEP: Run several restarts from frequency-based and crib-based keys; keep the
  key with highest fitness.
  JUSTIFICATION: Hill climbing can stop at a local maximum; multiple restarts
  increase the chance of finding the true key.

  Initial fitness (log-probability sum): -24710.75
  STEP: Iteratively swap two letters in the key and re-score decrypted text.
  JUSTIFICATION: If the new key yields higher n-gram fitness, we keep it. This
  escapes local errors from monogram-only mapping (e.g. similar-frequency
  letters).

  Iteration 0: fitness improved to -24413.63 (swap key[u]<->key[d])
  Iteration 16: fitness improved to -24338.60 (swap key[o]<->key[s])
  Iteration 34: fitness improved to -24264.19 (swap key[l]<->key[s])
  Iteration 35: fitness improved to -24247.82 (swap key[g]<->key[w])
  Iteration 63: fitness improved to -24229.49 (swap key[g]<->key[u])
  Iteration 123: fitness improved to -24227.94 (swap key[m]<->key[v])
  Iteration 215: fitness improved to -24211.24 (swap key[h]<->key[l])
  Iteration 275: fitness improved to -24180.50 (swap key[h]<->key[a])
  Iteration 317: fitness improved to -24176.92 (swap key[r]<->key[a])
  Iteration 479: fitness improved to -24155.21 (swap key[g]<->key[l])
  Iteration 483: fitness improved to -23923.54 (swap key[i]<->key[r])
  Iteration 625: fitness improved to -23921.11 (swap key[g]<->key[i])
  No improvement for 1500 iterations; stopping.
  Restart 1 (frequency-based) final fitness: -23921.11
  Restart 2 (crib meh->the) final fitness: -22974.51
  Restart 3 (frequency-based) final fitness: -24100.22
  Restart 4 (crib meh->the) final fitness: -22977.02
  Restart 5 (frequency-based) final fitness: -24020.09

============================================================
  Exhaustive pair-swap refinement
============================================================
  STEP: Try all 325 pair swaps in the key; keep any swap that improves n-gram
  fitness.
  JUSTIFICATION: This fixes remaining letter confusions (e.g. b/w, p/k) so the
  decrypted text is fully coherent.
```

6.2 Manual phase: letter-swap corrections and final key

The manual letter-swap corrections were applied in this order, with the reason for each: (1) (1, 22) b↔w (world/was); (2) (10, 21) k↔v (marketing/governments); (3) (1, 15) b↔p (product/described); (4) (9, 23) j↔x (adjacently); (5) (16, 23) q↔x (complex); (6) (16, 25) q↔z (blitzkrieg, emphasized). The log for the manual phase and the final key and decryption is given below. The key and file paths shown here match those presented in Section 5.

```
============================================================
  Manual letter-swap corrections for coherence
============================================================
  STEP: Apply swaps (b<->w), (k<->v), (b<->p), (j<->x), (q<->x), (q<->z) for
  full coherence.
  JUSTIFICATION: Fixes world/was, marketing/governments, product/described,
  adjacently, complex, blitzkrieg, emphasized.

  (1,22) b<->w: corrects 'world' vs 'was' (e.g. 'end of world war i')
  (10,21) k<->v: corrects 'marketing' vs 'governments' (e.g. 'military and
      government services')
  (1,15) b<->p: corrects 'product' vs 'described' (e.g. 'finished product', '
      described in detail')
  (9,23) j<->x: corrects 'adjacently' (e.g. 'used ... adjacently')
  (16,23) q<->x: corrects 'complex' (e.g. 'the most complex')
  (16,25) q<->z: corrects 'blitzkrieg' and 'emphasized'
  Applied swaps: (1,22)(10,21)(1,15)(9,23)(16,23)(16,25)

============================================================
  Final substitution key and decryption
============================================================
  STEP: Apply inverse substitution to ciphertext to obtain plaintext.
  JUSTIFICATION: Each cipher letter is replaced by the corresponding plain
  letter from the discovered key.

  Key (plain -> cipher):
  Plaintext  a b c d e f g h i j k l m n o p q r s t u v w x y z
  Ciphertext g t q d h b j e i v z r c w s p f x u m k o l y n a

  Decrypted text saved to: .../data/task1/article_decrypted.txt
  Key saved to: .../data/task1/substitution_key.txt
```

7 Observations and discussion

This section discusses observations from the decryption process and compares the approach with the SEED Lab exercise. The discussion draws on the methodology (Section 3), the results (Section 5), and the execution log (Section 6).

7.1 Observations

Ciphertext format and encoding scope. Inspection of the provided ciphertext and the decrypted output shows that **only the 26 letters (a–z) were encoded** by the substitution; all other characters are left unchanged by the decryption, consistent with the implementation. The ciphertext contains **no space characters**, **no digits**, and **no punctuation**. Numerals (e.g. years such as 1918 or 1923) do not appear in the ciphertext; they were likely omitted from the source before encryption or were not encoded—they are not present in the file and were not replaced by `[]` or any other symbol. The only non-letter characters present are the literal two-character sequence `[]` (in three positions) and newline characters; `[]` appears in the same positions in both ciphertext and decrypted text because the substitution is applied only to letters and those characters are preserved unchanged. Spaces were not encoded and are absent; the decrypted plaintext therefore has no word boundaries, and words run together (e.g. `theenigmamachinewasinventedby...`). **Case** is uniformly lowercase. These observations are consistent with a monoalphabetic substitution restricted to the 26 letters, with non-letters (`[]` and newlines) preserved and spaces and numerals omitted from the ciphertext. They also justify the application of frequency analysis to a continuous letter stream, which is standard for this form of cryptanalysis [2, 3].

Heuristic and search behaviour.

- **Crib versus frequency-based start.** Using the most frequent cipher trigram (`meh`) as `the` yielded consistently higher final fitness than the purely frequency-based initial key, consistent with `the` being the dominant English trigram [2, 4].
- **Hill climbing.** Fitness improved in discrete steps (e.g. from approximately −24710 to approximately −22974 on the best run). Random pair swaps corrected letter confusions (e.g. similar-frequency letters). Termination after 1500 iterations without improvement avoided prolonged runs without gain.

- **Exhaustive refinement.** After hill climbing, exhaustive pair-swap refinement did not alter the key in the run shown in Section 6; the best key was already at a local maximum over the 325 possible single swaps. Manual swaps were still required for a small number of letter positions (e.g. b/w, k/v) that the n-gram model did not discriminate sufficiently [3].
- **Reference data.** The provided JSON files contained 26 letters, 42 bigrams, and 16 trigrams. Sparse trigram coverage is mitigated by the floor probability for unknown n-grams; richer reference data would be expected to improve discrimination.

7.2 Comparison with the SEED Lab exercise

The coursework asks whether the approach to discovering the encryption key was the same as in the lab exercise.

No. The SEED Lab on Secret-Key Encryption [6] focuses on the *use* of secret-key encryption: algorithms, modes of operation (e.g. ECB, CBC, CFB, OFB), padding, and initialisation vectors, using tools and programs to encrypt and decrypt when the *key is already known*. It does not address *cryptanalysis* of a substitution cipher or the *recovery* of an unknown key.

In Task 1, the key was *unknown* and the objective was to *recover* it from the ciphertext alone. The approach was therefore cryptanalytic: n-gram frequency analysis (monograms, bigrams, trigrams), a fitness function, and search (hill climbing with restarts and exhaustive refinement) were used to find the substitution key that yields English-like plaintext. The methods thus differ: the lab concerns the correct use of encryption and decryption with a given key, whereas Task 1 concerns the cryptanalysis of a classical cipher using language statistics and search. Both form part of the same coursework but serve different learning objectives—encryption versus cryptanalysis.

8 Conclusion

This report has presented the approach, implementation, and results for breaking the substitution cipher in Task 1. In summary, the cipher was broken by: (i) loading letter, bigram, and trigram frequencies from the provided JSON data; (ii) constructing an initial key from cipher letter counts matched to English letter frequency order, and optionally from a crib (`meh` → `the`); (iii) maximising an n-gram log-probability fitness via hill climbing with random pair swaps and multiple restarts; (iv) exhaustive pair-swap refinement; and (v) a small number of manual letter swaps for coherence. The encryption key was recovered and the ciphertext decrypted to readable plaintext about the Enigma machine. It is concluded that monoalphabetic substitution is vulnerable to statistical attack when sufficient ciphertext and language-specific n-gram data are available [1, 3], and that the combination of frequency-based initial keys, cribs, and local search (hill climbing with restarts) constitutes an effective and well-established cryptanalytic approach [2, 3].

Part II

Task 4: Block Cipher Modes and Padding

9 Introduction

This section addresses Task 4 of COMP3028 Coursework 1: the encryption of the three plaintext files specified in the coursework (5, 10, and 16 bytes) using **AES-128** in four modes of operation—**ECB**, **CBC**, **CFB**, and **OFB**. The objectives are to identify which modes employ padding, to present the resulting ciphertext lengths, and to explain the rationale for padding in block-oriented modes and its absence in stream-oriented modes.

10 Methodology and Experimental Setup

10.1 Alignment with SEED Lab

The procedure follows the **SEED Lab (Secret-Key Encryption)** and its lab manual (Crypto_Encryption, seedsecuritylabs.org). Those materials specify the use of **OpenSSL** for AES-128 in ECB, CBC, CFB, and OFB modes, and describe PKCS#5/PKCS#7 padding for block modes and the absence of padding for CFB and OFB. The EVP sample code in the Labsetup is consistent with this behaviour.

10.2 Inputs and Parameters

- **Input files:** The three plaintext files specified in the coursework: `data/task4/f1.txt` (5 bytes), `f2.txt` (10 bytes), `f3.txt` (16 bytes).
- **Key and IV:** From `key_iv.txt` (Moodle): 16-byte key `00112233445566778899aabbccddeeff`, IV `0102030405060708090000a0b0c0d0e0f`.
- **Modes:** AES-128 in ECB (no IV), CBC, CFB, and OFB, with the same key and, where applicable, the same IV.

10.3 Encryption Procedure

Encryption was performed using **OpenSSL** via the script `scripts/encrypt_task4_openssl.py` when available, which invokes the following commands (as in the lab manual):

- **ECB:** `openssl enc -aes-128-ecb -e -K <key> -in <file> -out <file>`
- **CBC:** `openssl enc -aes-128-cbc -e -K <key> -iv <iv> -in <file> -out <file>`
- **CFB:** `openssl enc -aes-128-cfb -e -K <key> -iv <iv> -in <file> -out <file>`
- **OFB:** `openssl enc -aes-128-ofb -e -K <key> -iv <iv> -in <file> -out <file>`

OpenSSL was used from the system PATH when available (e.g. within the SEED Lab Docker environment or a local installation). The script was executed from the project root: `python scripts/encrypt_task4_openssl.py` or `encrypt_task4.py`.

An alternative implementation, `scripts/encrypt_task4.py`, uses Python with the PyCryptodome library when OpenSSL is not in PATH, replicating the same padding behaviour (PKCS#7 for ECB/CBC; no padding for CFB/OFB). Both implementations produce identical ciphertext lengths and padding behaviour.

10.4 Experimental Setup

Parameter	**Value**
Cipher	AES-128 (block size 16 bytes)
Key	From `key_iv.txt` (Moodle): `00112233445566778899aabbccddeeff`
IV	`0102030405060708090000a0b0c0d0e0f`
Input files	`data/task4/f1.txt`, `f2.txt`, `f3.txt` (5, 10, 16 bytes)
Tool	PyCryptodome (Python); OpenSSL `openssl enc` when available

11 Implementation

11.1 OpenSSL invocation

(`scripts/encrypt_task4_openssl.py`, `run_openssl_enc`, lines 32–42.) The primary script invokes `openssl enc` via `subprocess.run`. The command is built with the mode (`-aes-128-ecb`, `-aes-128-cbc`, etc.), key, input and output paths; ECB omits the IV, while CBC, CFB, and OFB include it.

Listing 15: Invoking OpenSSL for encryption.

```
def run_openssl_enc(mode: str, infile: Path, outfile: Path, key_hex: str, iv_hex: str | None) -> int:
    cmd = [
        "openssl", "enc", f"-aes-128-{mode}", "-e",
        "-K", key_hex, "-in", str(infile), "-out", str(outfile)
    ]
    if iv_hex and mode != "ecb":
        cmd.extend(["-iv", iv_hex])
    r = subprocess.run(cmd, capture_output=True, text=True, timeout=10)
    if r.returncode != 0:
        raise RuntimeError(f"openssl failed: {r.stderr or r.stdout}")
    return outfile.stat().st_size
```

11.2 Padding behaviour in Python alternative

(`scripts/encrypt_task4.py`, `encrypt_ecb`, `encrypt_cbc`, `encrypt_cfb`, `encrypt_ofb`, lines 38–64.) The alternative script uses PyCryptodome. Block modes (ECB, CBC) call `pad(plaintext, AES.block_size)` before encryption; stream-like modes (CFB, OFB) encrypt the plaintext directly without padding. This mirrors OpenSSL's behaviour.

Listing 16: ECB/CBC use padding; CFB/OFB do not.

```
def encrypt_ecb(plaintext: bytes, key: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_ECB)
    padded = pad(plaintext, AES.block_size)
    return cipher.encrypt(padded)

def encrypt_cfb(plaintext: bytes, key: bytes, iv: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CFB, iv=iv, segment_size=128)
    return cipher.encrypt(plaintext)
```

12 Results: Ciphertext Lengths

The following presents the ciphertext lengths observed for each mode and file size.

12.1 Summary Table

File	Plaintext length	ECB	CBC	CFB	OFB
f1.txt	5 bytes	16	16	5	5
f2.txt	10 bytes	16	16	10	10
f3.txt	16 bytes	32	32	16	16

12.2 Observations

- **ECB and CBC:** Ciphertext length is always a multiple of the block size (16 bytes) [7]. For `f3.txt`, which is exactly 16 bytes, PKCS#7 still appends a full padding block (16 bytes of value 0x10), yielding a ciphertext of 32 bytes. NIST SP 800-38A (Appendix A, p. 17) recommends padding every message, including those whose final block is already complete, by appending an entire block of padding [7, Appendix A]. The Handbook of Applied Cryptography similarly states that ECB and CBC require "n-bit plaintext blocks" and references Algorithm 9.58 for padding [8, Ch. 7, Sect. 7.2.2].
- **CFB and OFB:** Ciphertext length equals plaintext length; no padding is applied [7, 1]. NIST specifies that for OFB (and CTR), "the plaintext need not be a multiple of the block size" and the last ciphertext block may be a partial block [7, Sect. 5.2]; CFB with segment sizes smaller than the block size similarly produces ciphertext equal in length to the plaintext. Stallings explains that CFB and OFB "turn a block cipher into a stream cipher" and thus operate without requiring block alignment [1, Ch. 6].

12.3 Verification

The script was executed from the project root: `python scripts/encrypt_task4.py` (or `encrypt_task4_openssl.py` when OpenSSL is in PATH). The representative output below was produced by the script and confirms the ciphertext lengths in the Summary Table. When OpenSSL is not available, `encrypt_task4.py` uses PyCryptodome and yields identical results; the methodology and padding behaviour are unchanged.

```
Task 4: Padding - AES-128 ECB, CBC, CFB, OFB
Key (hex): 00112233445566778899aabbccddeeff
IV  (hex): 0102030405060708090000a0b0c0d0e0f

File: f1.txt  |  Plaintext length: 5 bytes  |  Expected: 5
  ECB  | Ciphertext length: 16 bytes | Hex: 3846c2a1c915c13fb0ed060622d7d022
  CBC  | Ciphertext length: 16 bytes | Hex: 665d8a869d00763a20e0ac01b5aeb6ef
  CFB  | Ciphertext length: 5 bytes | Hex: ea0135a438
  OFB  | Ciphertext length: 5 bytes | Hex: ea0135a438
```

13 Explanation: Why Some Modes Require Padding and Others Do Not

The differing ciphertext lengths reflect a fundamental distinction between block-oriented and stream-oriented modes. ECB and CBC use padding; CFB and OFB do not. The following subsections explain the rationale.

Modes **with** padding	Modes **without** padding
ECB, CBC	CFB, OFB

13.1 Modes That Use Padding: ECB and CBC

ECB and CBC are **block cipher modes**: they process the plaintext in **fixed-size blocks** (for AES, 16 bytes per block) [7, 1]. NIST SP 800-38A specifies that "for the ECB and CBC modes, the total number of bits in the plaintext must be a multiple of the block size" [7, Sect. 5.2]. The block cipher primitive accepts only inputs of exactly one block length. When the plaintext length is not a multiple of the block size, the final block is incomplete, so **padding** (e.g. PKCS#7) is appended to form a whole number of blocks before encryption [8, 6]. The Handbook of Applied Cryptography notes that ECB and CBC require "n-bit plaintext blocks" and references Algorithm 9.58 for padding [8, Ch. 7, Sect. 7.2.2]. Stallings' textbook similarly describes ECB and CBC as operating on fixed block sizes [1, Ch. 6].

Padding is necessary because the cipher operates on complete blocks only; any shorter input must be extended to a full block. On decryption, the padding is removed according to a well-defined scheme (e.g. the last byte encodes the number of padding bytes).

13.2 Modes That Do Not Use Padding: CFB and OFB

CFB (Cipher Feedback) and OFB (Output Feedback) are **stream-cipher-like** modes: they use the block cipher to generate a **keystream**, which is then XORed with the plaintext [7, 1]. NIST SP 800-38A states that for OFB and CTR, "the plaintext need not be a multiple of the block size" and the ciphertext may include a partial final block [7, Sect. 5.2]; thus the ciphertext length equals the plaintext length.

No padding is needed because encryption and decryption use the same keystream and the XOR is applied byte-by-byte (or segment-by-segment). Only as many keystream bytes as there are plaintext bytes are consumed; no additional bytes are required [8, 6]. The Handbook of Applied Cryptography describes CFB and OFB as processing "r-bit blocks" (where $r \leq n$)

without requiring full block alignment [8, Ch. 7, Sect. 7.2.2]. Katz and Lindell note that stream-cipher modes allow encryption of arbitrary-length messages without padding [9, Ch. 3].

14 Reference Commands

The following OpenSSL commands were used, consistent with the SEED Lab manual:

```
# ECB (no IV)
openssl enc -aes-128-ecb -e -K <key_hex> -in f1.txt -out f1_ecb.bin

# CBC, CFB, OFB (with IV)
openssl enc -aes-128-cbc -e -K <key_hex> -iv <iv_hex> -in f1.txt -out f1_cbc.bin
openssl enc -aes-128-cfb -e -K <key_hex> -iv <iv_hex> -in f1.txt -out f1_cfb.bin
openssl enc -aes-128-ofb -e -K <key_hex> -iv <iv_hex> -in f1.txt -out f1_ofb.bin
```

OpenSSL applies PKCS#7 padding for ECB and CBC; CFB and OFB are used without padding, as documented in the SEED Lab manual [6].

15 Deliverables

Item	Description
Primary script	`scripts/encrypt_task4_openssl.py` — OpenSSL-based encryption for all four modes
Alternative script	`scripts/encrypt_task4.py` — Python/PyCryptodome implementation with equivalent padding behaviour
Ciphertexts	`data/task4/f1_ecb.bin`, `f1_cbc.bin`, `f1_cfb.bin`, `f1_ofb.bin`, and corresponding files for `f2` and `f3`

Verification was performed by inspection of the script output and, where applicable, hex dumps of the ciphertext files.

Part III

Task 5: Error Propagation

16 Introduction

This section addresses Task 5 of COMP3028 Coursework 1: to encrypt the designated plaintext with AES-128 in ECB, CBC, CFB, and OFB modes; to corrupt a single bit of the 55th byte in each resulting ciphertext; to decrypt the corrupted ciphertexts with the same key and IV; and to report how much information can be recovered in each mode, with justification. The procedure follows the coursework specification and the SEED Lab (Secret-Key Encryption) and its lab manual (Crypto_Encryption, Section 7, Error Propagation).

17 Methodology and Experimental Setup

17.1 Alignment with the SEED Lab

The SEED Lab manual (Crypto_Encryption, seedsecuritylabs.org, Section 7) prescribes: (1) encrypt a file of at least 1000 bytes with AES-128; (2) corrupt one bit of the 55th byte in the encrypted file (e.g. using a hex editor); (3) decrypt the corrupted ciphertext with the correct key and IV; (4) analyse how much information is recoverable for ECB, CBC, CFB, and OFB. The implementation described below adheres to this procedure. Encryption and decryption were performed either with OpenSSL (`openssl enc`) when available, or with the PyCryptodome library under Python, both yielding identical behaviour for the purposes of this task.

17.2 Inputs and Parameters

- **Plaintext:** The Task 5 plaintext file specified in the coursework (`data/task5/task5_plaintext.txt`), length 1078 bytes.
- **Key and IV:** From `key_iv.txt` (Moodle): 16-byte key `00112233445566778899aabbccddeeff`, IV `0102030405060708090000a0b0c0d0e0f`.
- **Corruption:** Exactly one bit of the 55th byte (0-based index 54) of each ciphertext was flipped (XOR with 0x01, least significant bit).
- **Modes:** AES-128 in ECB (no IV), CBC, CFB, and OFB, with the same key and, where applicable, the same IV.

17.3 Encryption and Decryption Procedure

Encryption and decryption were performed using **OpenSSL** via the script `scripts/task5_error_propagation.py`, which invokes the following commands (as in the SEED Lab manual, consistent with Task 4):

- **Encrypt (ECB):** `openssl enc -aes-128-ecb -e -K <key> -in <file> -out <file>`
- **Encrypt (CBC, CFB, OFB):** `openssl enc -aes-128-<mode> -e -K <key> -iv <iv> -in <file> -out <file>`
- **Decrypt (ECB):** `openssl enc -aes-128-ecb -d -K <key> -in <file> -out <file>`
- **Decrypt (CBC, CFB, OFB):** `openssl enc -aes-128-<mode> -d -K <key> -iv <iv> -in <file> -out <file>`

OpenSSL was used from the system PATH when available. The script was executed from the project root: `python scripts/task5_error_propagation.py`.

An alternative implementation uses the PyCryptodome library when OpenSSL is not in PATH, replicating the same padding behaviour as Task 4 (PKCS#7 for ECB/CBC; no padding for CFB/OFB).

17.4 Experimental Setup

Parameter	Value
Cipher	AES-128 (block size 16 bytes)
Key	From `key_iv.txt` (Moodle): `00112233445566778899aabbccddeeff`
IV	`010203040506070809000a0b0c0d0e0f`
Plaintext	`data/task5/task5_plaintext.txt` (1078 bytes)
Corruption	One bit flipped at byte index 54 (55th byte), mask 0x01
Tool	PyCryptodome (Python); OpenSSL `openssl enc` when available

The script produced, for each mode, an encrypted file, a corrupted ciphertext (one bit flipped at byte index 54), and the decryption of the corrupted ciphertext. Outputs are written to `data/task5/` with the naming convention `task5_<mode>.bin`, `task5_<mode>_corrupted.bin`, and `task5_<mode>_decrypted.bin`.

18 Implementation

18.1 OpenSSL invocation

(`scripts/task5_error_propagation.py`, `run_openssl_enc`, `run_openssl_dec`, lines 45–71.) The script invokes `openssl enc` via `subprocess.run` for both encryption and decryption. The command is built with the mode (`-aes-128-ecb`, `-aes-128-cbc`, etc.), `-e` or `-d`, key, input and output paths; ECB omits the IV, while CBC, CFB, and OFB include it.

Listing 17: Invoking OpenSSL for encryption and decryption.

```python
def run_openssl_enc(mode: str, infile: Path, outfile: Path, key_hex: str, iv_hex: str | None) -> None:
    cmd = [
        "openssl", "enc", f"-aes-128-{mode}", "-e",
        "-K", key_hex, "-in", str(infile), "-out", str(outfile)
    ]
    if iv_hex and mode != "ecb":
        cmd.extend(["-iv", iv_hex])
    r = subprocess.run(cmd, capture_output=True, text=True, timeout=30)
    if r.returncode != 0:
        raise RuntimeError(f"openssl enc failed: {r.stderr or r.stdout}")

def run_openssl_dec(mode: str, infile: Path, outfile: Path, key_hex: str, iv_hex: str | None) -> None:
    cmd = [
        "openssl", "enc", f"-aes-128-{mode}", "-d",
        "-K", key_hex, "-in", str(infile), "-out", str(outfile)
    ]
    if iv_hex and mode != "ecb":
        cmd.extend(["-iv", iv_hex])
    r = subprocess.run(cmd, capture_output=True, text=True, timeout=30)
    if r.returncode != 0:
        raise RuntimeError(f"openssl dec failed: {r.stderr or r.stdout}")
```

18.2 Padding behaviour in Python alternative

(`scripts/task5_error_propagation.py`, `_encrypt_decrypt_pycrypto`, lines 76–103.) When OpenSSL is not in PATH, the script uses PyCryptodome. Block modes (ECB, CBC) call `pad(plaintext, AES.block_size)` before encryption and `unpad` after decryption; stream-like modes (CFB, OFB) encrypt and decrypt the plaintext directly without padding. This mirrors OpenSSL's behaviour and is consistent with Task 4.

18.3 One-bit corruption

(`scripts/task5_error_propagation.py`, lines 29–31, 106–113.) The 55th byte (1-based) corresponds to index 54 (0-based). A single bit is flipped by XORing that byte with `0x01` (least significant bit). The `corrupt_one_bit` function returns a copy of the ciphertext with the specified bit toggled.

Listing 18: Corruption parameters and one-bit flip.

```
# 55th byte (1-based) = index 54 (0-based). Corrupt exactly one bit (LSB flip).
BYTE_INDEX_TO_CORRUPT = 54
BIT_MASK = 0x01  # flip LSB

def corrupt_one_bit(data: bytes, byte_index: int, bit_mask: int = 0x01) -> bytes:
    arr = bytearray(data)
    arr[byte_index] ^= bit_mask
    return bytes(arr)
```

18.4 Recovery comparison

(`scripts/task5_error_propagation.py`, `compare_recovery`, lines 116–139.) The script compares the original plaintext with the decrypted (possibly corrupted) output byte-by-byte. It reports the number of matching bytes, the first index at which they differ, and the recoverable percentage. Extra bytes in the decrypted output (e.g. from padding in ECB/CBC) are not counted as recovered.

Listing 19: Comparing original and decrypted plaintext.

```
def compare_recovery(original: bytes, decrypted: bytes) -> dict:
    n_orig, n_dec = len(original), len(decrypted)
    n_compare = min(n_orig, n_dec)
    matches = sum(1 for i in range(n_compare) if original[i] == decrypted[i])
    first_error = None
    for i in range(n_compare):
        if original[i] != decrypted[i]:
            first_error = i
            break
    pct = (100.0 * matches / n_orig) if n_orig else 0.0
    return {"matching_bytes": matches, "first_error_index": first_error, "recoverable_pct": pct, ...}
```

19 Results: Recovery After One-Bit Corruption

The following presents the recovery statistics for each mode after corrupting one bit of the 55th ciphertext byte.

19.1 Summary of Recovered Information

Table 2 summarises, for each mode, the ciphertext length, the number of plaintext bytes that match the original after decrypting the corrupted ciphertext, and the byte index at which the first error appears.

Mode	Ciphertext length	Recovered bytes (of 1078)	First error (byte index)
ECB	1088	1062 (98.52%)	48
CBC	1088	1062 (98.52%)	48
CFB	1078	1061 (98.42%)	54
OFB	1078	1077 (99.91%)	54

Table 2: Recovery statistics after corrupting one bit of the 55th ciphertext byte.

19.2 Interpretation by Mode

The observed recovery patterns match the error-propagation properties documented in NIST SP 800-38A (Appendix D, p. 21) [7, Appendix D], the Handbook of Applied Cryptography [8, Ch. 7, Sect. 7.2.2], and Stallings' textbook [1, Ch. 6]:

- **ECB.** The 55th ciphertext byte lies in the block spanning bytes 48–63 (block index 3). NIST states that for ECB, "bit errors within a ciphertext block do not affect the decryption of any other blocks" [7, Appendix D]. The Handbook of Applied Cryptography similarly notes that "one or more bit errors in a single ciphertext block affect decipherment of that block only" [8, Alg. 7.11, p. 226]. Only that block is affected; all other blocks decrypt correctly. Thus bytes 0–47 and 64–1077 are recovered ($48 + 1014 = 1062$ bytes). The first error occurs at byte index 48.

- **CBC.** The corrupted ciphertext block corrupts the corresponding plaintext block (bytes 48–63). In CBC decryption, $P_i = D(C_i) \oplus C_{i-1}$, so the corrupted C_3 causes block 3 to be fully corrupted (avalanche effect). NIST states that "any bit positions that contain bit errors in a ciphertext block will also contain bit errors in the decryption of the succeeding ciphertext block; the other bit positions are not affected" [7, Appendix D]. The Handbook notes that "the recovered plaintext x'_{j+1} has bit errors precisely where c_j did" [8, Alg. 7.13, p. 231]. The same corrupted C_3 is XORed when decrypting block 4, so exactly one bit of block 4 is wrong (in the same bit position). Typically 17 bytes are corrupted; in our run, one byte in block 3 matched the original by chance, yielding 16 errors and 1062 bytes recovered. The first error is at byte 48.

- **CFB.** In CFB (stream-like), the corrupted ciphertext byte affects the corresponding plaintext byte (index 54) directly. NIST states that "bit errors in a ciphertext segment affect the decryption of the next b/s (rounded up) ciphertext segments" [7, Appendix D]. The same corrupted byte is fed into the feedback path, so the following segment (next block) is corrupted as well [1, 6]. Stallings describes CFB error propagation as affecting the current and following segments [1, Ch. 6]. The first error is at byte 54; 17 bytes are incorrect in total (1061 recovered).

- **OFB.** NIST states that for OFB, "bit errors within a ciphertext block do not affect the decryption of any other blocks" and "the bit error(s) in the decrypted ciphertext block occur in the same bit position(s) as in the ciphertext block" [7, Appendix D]. The keystream is generated from the key and IV only; it does not depend on the ciphertext [8, Ch. 7]. Corrupting one ciphertext byte therefore affects only the corresponding plaintext byte (index 54). No propagation occurs; 1077 bytes are recovered and a single byte is wrong.

19.3 Justification

These results are consistent with the SEED Lab manual (Section 7) [6] and with the standard error-propagation properties of the four modes defined in NIST SP 800-38A (Appendix D: Error Properties, p. 21) [7, Appendix D] and described in standard textbooks [1, 8]: block modes (ECB, CBC) exhibit block-level or chained corruption, while stream-like modes (CFB, OFB) confine the effect to the affected ciphertext position and, in CFB, to the following segment. OFB shows the least propagation (one byte only), as expected from its definition. Table D.1 in NIST SP 800-38A summarises these effects [7, Appendix D]; the Handbook of Applied Cryptography provides the same characterisation in Section 7.2.2 [8, Ch. 7].

19.4 Verification

The script was executed from the project root: `python scripts/task5_error_propagation.py`. The output below was produced by the script and confirms the recovery statistics in Table 2. When OpenSSL is not in PATH, the script uses PyCryptodome and yields identical results; the methodology and padding behaviour are unchanged. The reported counts and first-error indices were confirmed by byte-wise comparison of the decrypted files against the original plaintext.

```
Task 5: Error Propagation - Corrupted Cipher Text
Key (hex): 00112233445566778899aabbccddeeff
IV  (hex): 0102030405060708090000a0b0c0d0e0f
Plaintext: .../task5_plaintext.txt (1078 bytes)
Corruption: flip 1 bit (mask=0x01) at byte index 54 (55th byte)

  ECB | cipher: 1088 bytes | recovered: 1062/1078 (98.52%) | first error at byte 48
  CBC | cipher: 1088 bytes | recovered: 1062/1078 (98.52%) | first error at byte 48
  CFB | cipher: 1078 bytes | recovered: 1061/1078 (98.42%) | first error at byte 54
  OFB | cipher: 1078 bytes | recovered: 1077/1078 (99.91%) | first error at byte 54
```

20 Reference Commands

The following OpenSSL commands were used, consistent with the SEED Lab manual (Section 7) and Task 4:

```
# Encrypt (ECB, no IV)
openssl enc -aes-128-ecb -e -K <key_hex> -in task5_plaintext.txt -out task5_ecb.bin

# Encrypt (CBC, CFB, OFB with IV)
openssl enc -aes-128-cbc -e -K <key_hex> -iv <iv_hex> -in task5_plaintext.txt -out task5_cbc.bin
openssl enc -aes-128-cfb -e -K <key_hex> -iv <iv_hex> -in task5_plaintext.txt -out task5_cfb.bin
openssl enc -aes-128-ofb -e -K <key_hex> -iv <iv_hex> -in task5_plaintext.txt -out task5_ofb.bin
```

```

# Decrypt corrupted ciphertext (example: CBC)
openssl enc -aes-128-cbc -d -K <key_hex> -iv <iv_hex> -in task5_cbc_corrupted.bin -out
    task5_cbc_decrypted.bin
```

OpenSSL applies PKCS#7 padding for ECB and CBC; CFB and OFB are used without padding, as documented in the SEED Lab manual [6].

21 Deliverables

Item	**Description**
Script	`scripts/task5_error_propagation.py` — OpenSSL-based encryption/decryption when available; PyCryptodome fallback with equivalent padding behaviour
Encrypted files	`data/task5/task5_ecb.bin`, `task5_cbc.bin`, `task5_cfb.bin`, `task5_ofb.bin`
Corrupted ciphertexts	`data/task5/task5_<mode>_corrupted.bin` for each mode
Decrypted outputs	`data/task5/task5_<mode>_decrypted.bin` for each mode

Verification was performed by inspection of the script output and byte-wise comparison of the decrypted files against the original plaintext.

References

[1] W. Stallings, *Cryptography and Network Security: Principles and Practice.* Boston: Pearson, 8th ed., 2020. Ch. 6: Block cipher modes ECB, CBC, CFB, OFB; ECB block independence; CBC/CFB error propagation; stream-cipher-like behaviour of CFB/OFB; padding requirements for block modes.

[2] J. Kun, “Cryptanalysis with N-grams.” Math & Programming, 2012. Accessed for this report. Discusses substitution cipher representation, trigram and bigram scoring, steepest ascent, and bigram-guided neighbour generation.

[3] Practical Cryptography, “Cryptanalysis of the simple substitution cipher.” Practical Cryptography, n.d. Describes hill climbing with quadgram fitness, random key and swap steps, multiple restarts, unicity distance, and limitations.

[4] P. Norvig, “Letter n-gram counts.” count_2l.txt, count_3l.txt, n.d. Widely used for bigram/trigram statistics; referenced in Kun (2012).

[5] Code Review Stack Exchange, “Decrypting a substitution cipher using n-gram frequency analysis.” Stack Exchange, n.d. Discussion and code for n-gram-based decryption.

[6] SEED Labs, “Secret-key encryption.” SEED Labs 2.0 – Crypto Lab, 2020. Lab on encryption algorithms, modes, padding, and IV; uses tools and programs to encrypt/decrypt messages.

[7] M. Dworkin, “Recommendation for block cipher modes of operation: Methods and techniques,” Tech. Rep. SP 800-38A, National Institute of Standards and Technology, 2001. Appendix A (p. 17): Padding for ECB/CBC/CFB; full-block padding when plaintext already complete. Sect. 5.2: OFB/CTR allow partial blocks. Appendix D (p. 21): Error properties—ECB/OFB/CTR: no cross-block propagation; CBC: bit errors propagate to succeeding block; CFB: errors affect next b/s segments.

[8] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* Boca Raton: CRC Press, 1996. Ch. 7, Sect. 7.2.2 (pp. 230–236): ECB (Alg. 7.11)—error affects that block only; CBC (Alg. 7.13)—single bit error affects blocks cj and cj+1, recovered plaintext has bit errors precisely where ciphertext did; CFB/OFB stream-like; padding (Alg. 9.58) for ECB/CBC.

[9] J. Katz and Y. Lindell, *Introduction to Modern Cryptography.* Boca Raton: CRC Press, 3rd ed., 2020. Ch. 3: Modes of operation and encryption in practice; block-cipher vs. stream-cipher modes.