Classical Symmetric-Cipher Cryptanalysis in Practice:

Breaking Repeating-Key XOR and Detecting AES in ECB Mode (Cryptopals Set 1, Challenges 6 and 8)

July 12, 2026

Abstract

Two canonical exercises in applied cryptanalysis illustrate how *structure that survives encryption* is sufficient to defeat confidentiality, even without attacking the underlying cipher primitive. Working from the public Cryptopals Set 1 challenge files, this report documents complete, reproducible attacks on (i) a message enciphered with repeating-key XOR and then Base64-encoded (Challenge 6), and (ii) a corpus of hex-encoded ciphertexts, exactly one of which was produced by the Advanced Encryption Standard (AES) in Electronic Codebook (ECB) mode (Challenge 8). For Challenge 6 we estimate the key length using the averaged normalized bit-level Hamming distance between key-length-sized blocks, transpose the ciphertext into single-key-byte streams, and solve each stream by χ^2 frequency scoring against an English letter-and-space model. The procedure recovers a key of length **29** bytes, `Terminator X: Bring the noise`, and the full 2,876-byte plaintext (printable ratio 1.0000). The winning key length simultaneously minimizes both the normalized Hamming distance (2.7471 bits/byte) and the mean per-column χ^2 (40.93 versus >600 for every competitor), making the identification unambiguous. For Challenge 8 we exploit the deterministic, stateless, block-wise nature of ECB: identical 16-byte plaintext blocks encrypt to identical ciphertext blocks under a fixed key. Counting repeated 16-byte blocks over all 204 ciphertexts identifies line **133** as the unique ECB ciphertext—the only line exhibiting *any* block repetition (one block recurring four times; 7 unique of 10 blocks). We provide the runnable, dependency-free attack code and its verified output, and close with a discussion of why repeating-key XOR and ECB fail the modern (IND-CPA / semantic-security) standard, and how randomized and authenticated modes remedy the leaks.

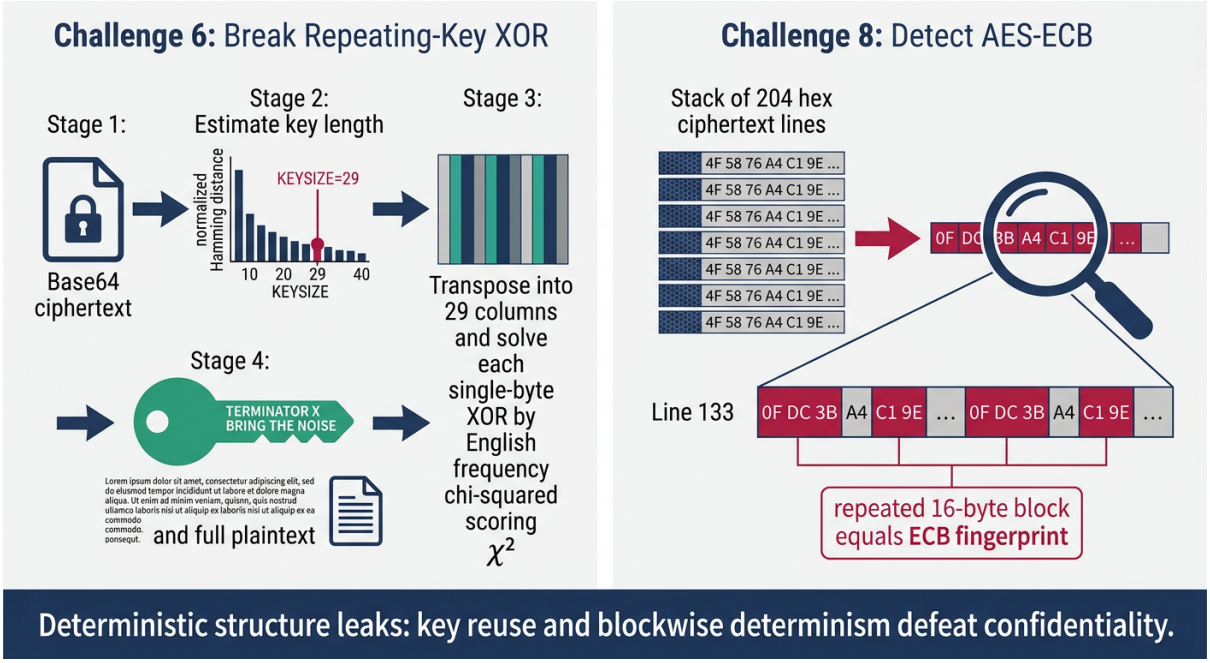

Figure 1. Graphical abstract. Left: the repeating-key XOR attack pipeline of Challenge 6—from the Base64 ciphertext, through Hamming-distance key-length estimation and column-wise single-byte-XOR frequency scoring, to the recovered key and plaintext. Right: the AES-ECB detector of Challenge 8, which flags line 133 by its repeated 16-byte ciphertext block. Both attacks succeed by reading residual structure, not by breaking the cipher primitive.

1 Introduction

Modern cryptography rests on a deceptively simple promise: a ciphertext should reveal nothing about its plaintext beyond the length. Formalized by Shannon's information-theoretic account of secrecy systems [1] and sharpened into the notions of semantic security and indistinguishability by Goldwasser and Micali [2], this promise is the yardstick against which every practical construction is measured. Cryptanalysis is the discipline of testing that promise—of asking whether some observable regularity of the ciphertext betrays the message or the key. The most instructive failures are rarely exotic breaks of the mathematical core of a cipher; far more often they are failures of *how* a strong primitive is used, in which ordinary statistical structure of real data leaks straight through the encryption layer. A cornerstone of the field, Kerckhoffs's principle [3], insists that a system's security must rest entirely on the secrecy of its key and not on the obscurity of its algorithm; the corollary, borne out by both attacks below, is that once the algorithm is known any residual structure in the ciphertext is fair game for the analyst.

The exercises studied here, drawn from Set 1 of the widely used Cryptopals challenges [4], are archetypes of this phenomenon. Neither attack requires factoring, lattice reduction, or differential trails [5]; each succeeds because the encryption scheme preserves a statistical or combinatorial invariant of the plaintext. Challenge 6 revisits a scheme that is, in essence, a byte-wise Vigenère cipher: a short key XORed repeatedly across the message. Such repeating-key schemes have been breakable since the nineteenth century, when Kasiski [6] and later Friedman [7] showed how to recover the key *period* from the ciphertext alone, after which each key position collapses to an elementary mono-alphabetic problem solvable by frequency analysis. The XOR variant enjoys perfect secrecy *only* in the limit of a truly random, never-reused, message-length key—the one-time pad of Vernam [8] and Shannon [1]. The moment the key is shorter than the message and therefore repeats, that guarantee collapses, and the ciphertext becomes a stack of interleaved single-byte-XOR problems.

Challenge 8 concerns a genuinely strong primitive—AES, the Rijndael cipher standardized as FIPS 197 [9, 10]—deployed in the weakest of the classical modes of operation, ECB. The AES block transform is not attacked at all. Instead, the attack exploits a property of the *mode*: because ECB encrypts each 16-byte block independently and deterministically, identical plaintext blocks always yield identical ciphertext blocks. The consequence, catalogued in the NIST guidance on block-cipher modes [11] and in comparative evaluations [12], is that any repetitive plaintext imprints a visible pattern on the ciphertext—the same defect that famously leaves the outline of an encrypted image intact. Detecting the ECB ciphertext in a mixed corpus therefore reduces to searching for repeated ciphertext blocks.

This report is organized around the two attacks. Section 2 describes the datasets and the reproducibility discipline. Section 3 develops the repeating-key XOR attack in full—bit-level Hamming distance (Section 3.2), key-length estimation, transposition, and per-position χ^2 frequency scoring (Section 3.3)—and reports the recovered key and plaintext (Section 3.4). Section 4 develops the AES-ECB detector, explaining ECB determinism (Section 4.1), the repeated-block criterion (Section 4.2), and the diagnostics that single out line 133 (Section 4.3). Section 5 interprets both results through the lens of modern provable security [13, 2] and derives the practical lesson: confidentiality demands randomized, and ideally authenticated, encryption [14, 15, 16]. Complete, runnable attack code and the full recovered plaintext appear in the appendices.

2 Materials and Methods

Datasets. The two inputs are the official Cryptopals Set 1 challenge files [4], retrieved over HTTPS from `https://cryptopals.com/static/challenge-data/6.txt` and `https://cryptopals.com/static/challenge-data/8.txt`. Their structure was validated before any

analysis (Table 1). The Challenge 6 file is a single Base64 blob (64 wrapped lines, 3,836 Base64 characters) that decodes to 2,876 raw ciphertext bytes. The Challenge 8 file is a list of 204 lines, each a 320-character hex string decoding to exactly 160 bytes—precisely ten 16-byte AES blocks per line. All lines passed strict Base64/hex validation, and SHA-256 digests of both the raw and decoded content were recorded to make the pipeline auditable.

Table 1. Structure and integrity of the input datasets (all checks passed).

Dataset	Raw bytes	Structure	Decoded
`6.txt` (Base64)	3,900	64 lines / 3,836 b64 chars	2,876 bytes
`8.txt` (hex)	65,484	204 lines / 320 hex chars each	160 bytes/line (= 10 blocks)

Implementation and reproducibility. Both attacks are implemented in pure Python 3 using only the standard library—no third-party cryptographic or numerical dependencies—so that every step is transparent and independently verifiable. The bit-level Hamming routine is self-tested against the canonical Cryptopals vector (`"this is a test"` vs. `"wokka wokka!!!"` = 37 bits) before use. The full attack sources are listed in Appendices A and B; the recovered plaintext is reproduced verbatim in Appendix D. The quantitative figures in this report (Figures 3–8) were regenerated directly from the challenge files and reproduce the reported values exactly, including the winning key length (29), the recovered key, and the detected line (133).

3 Challenge 6 — Breaking Repeating-Key XOR

3.1 The cipher and its weakness

Repeating-key XOR enciphers a plaintext byte stream $P = p_0 p_1 \dots p_{n-1}$ under a key $K = k_0 k_1 \dots k_{m-1}$ of length $m < n$ by the rule

$$c_i \;=\; p_i \oplus k_{\,i \bmod m}, \qquad 0 \le i < n, \tag{1}$$

where $\oplus$ is bit-wise exclusive-or. It is the byte-oriented incarnation of the Vigenère cipher and, equivalently, a one-time pad in which the pad has been fatally shortened and then cycled. Because $\oplus$ is its own inverse, decryption is identical to encryption; secrecy therefore depends entirely on the key. When $m = n$ and K is uniformly random and used once, Equation (1) is the Vernam one-time pad and achieves Shannon's perfect secrecy [8, 1]. When $m \ll n$, the same key byte k_j is reused on every plaintext byte whose index is congruent to j modulo m (Figure 2). Each such residue class is thus a *mono-alphabetic* single-byte-XOR cipher, and mono-alphabetic ciphers are trivially broken by frequency analysis [17, 18]. The attack therefore has two phases: recover m, then solve m independent single-byte problems.

3.2 Estimating the key length by normalized Hamming distance

The first phase estimates m without any knowledge of the plaintext. The key observation is statistical. The bit-level Hamming distance between two byte strings a and b of equal length is the population count of their XOR,

$$d_H(a, b) \;=\; \sum_i \mathrm{popcount}(a_i \oplus b_i), \tag{2}$$

i.e. the number of differing bits [19]. Now consider two blocks of ciphertext that are each exactly m bytes long and aligned to the key period. Under Equation (1) the two blocks were XORed with the *same* key material, so the XOR of the two ciphertext blocks equals the XOR of the

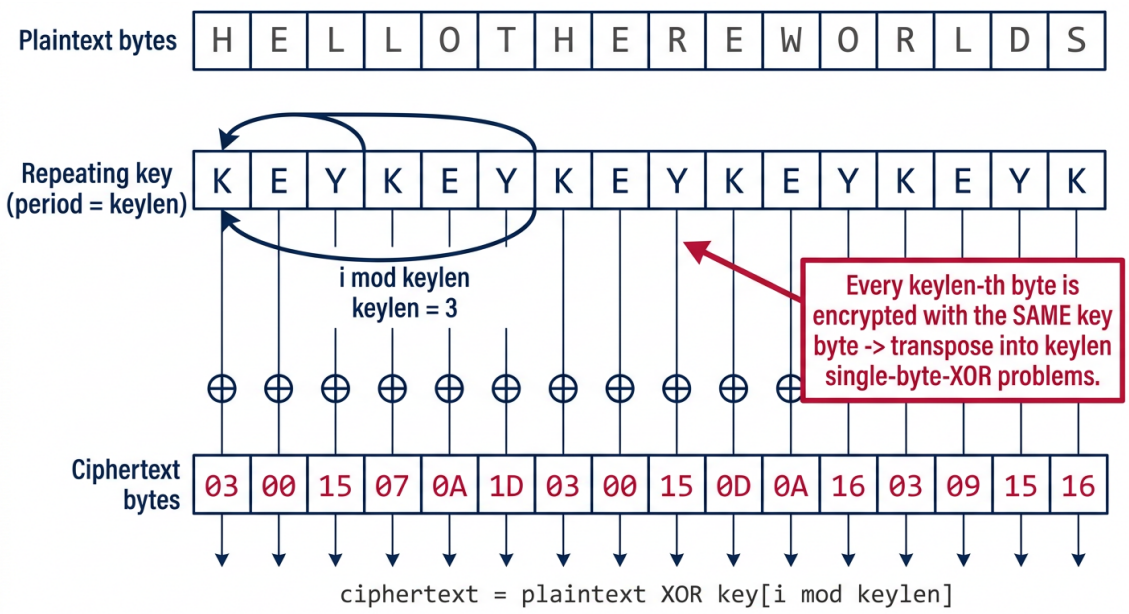

Figure 2. Repeating-key XOR (Equation (1)). The key cycles with period m, so every m-th ciphertext byte was produced with the *same* key byte. This is the structural flaw the attack exploits: the ciphertext decomposes into m interleaved single-byte-XOR ciphertexts.

two underlying plaintext blocks—the key cancels. English text has low per-bit entropy, so two plaintext fragments differ in relatively few bits; the normalized distance d_H/m is therefore *small* when m is the true key length (or a multiple of it) and larger for misaligned guesses, whose XOR mixes different key bytes and looks more random. This is the byte-XOR analogue of the Friedman/index-of-coincidence period test for classical Vigenère [7, 1].

Concretely, for each candidate length KEYSIZE $\in \{2, \ldots, 40\}$ we take the first four consecutive blocks $B_1, \ldots, B_4$ of that size and average the normalized Hamming distance over all $\binom{4}{2} = 6$ block pairs,

$$S(\text{KEYSIZE}) \;=\; \frac{1}{6} \sum_{1 \le p < q \le 4} \frac{d_H(B_p, B_q)}{\text{KEYSIZE}}. \tag{3}$$

Averaging over several pairs, rather than using a single pair, suppresses the sampling noise that otherwise makes short spurious key sizes look attractive. Candidate lengths are then ranked by ascending S. Figure 3 shows the resulting spectrum. The global minimum sits unambiguously at KEYSIZE $= 29$ with $S = 2.7471$ bits/byte; the runners-up (5, 2, 24, 7) all exceed 2.9. Small integer multiples and divisors of the true period often score well too, which is why the second phase confirms the choice by actually decrypting.

3.3 Transposition and per-position single-byte-XOR frequency scoring

Given a candidate length m, the second phase reorganizes the ciphertext so that the monoalphabetic structure becomes explicit. We *transpose* the byte stream into m columns: column j collects every byte whose index satisfies $i \equiv j \pmod m$, namely $c_j, c_{j+m}, c_{j+2m}, \ldots$ By construction each column was produced by XOR with the single key byte k_j (Figure 4). Recovering the key thus reduces to solving m independent single-byte-XOR ciphers, one per column.

Each column is solved by exhaustive search over the 256 possible key bytes, scoring the resulting candidate plaintext for "English-ness" with Pearson's χ^2 goodness-of-fit statistic [20]. Let n be the column length, let $E_c = f_c\, n$ be the expected count of character c under a reference English frequency model f_c (letters plus the space character, the single most common symbol in English prose; Figure 5), and let O_c be the observed count after XOR with a trial key byte. The

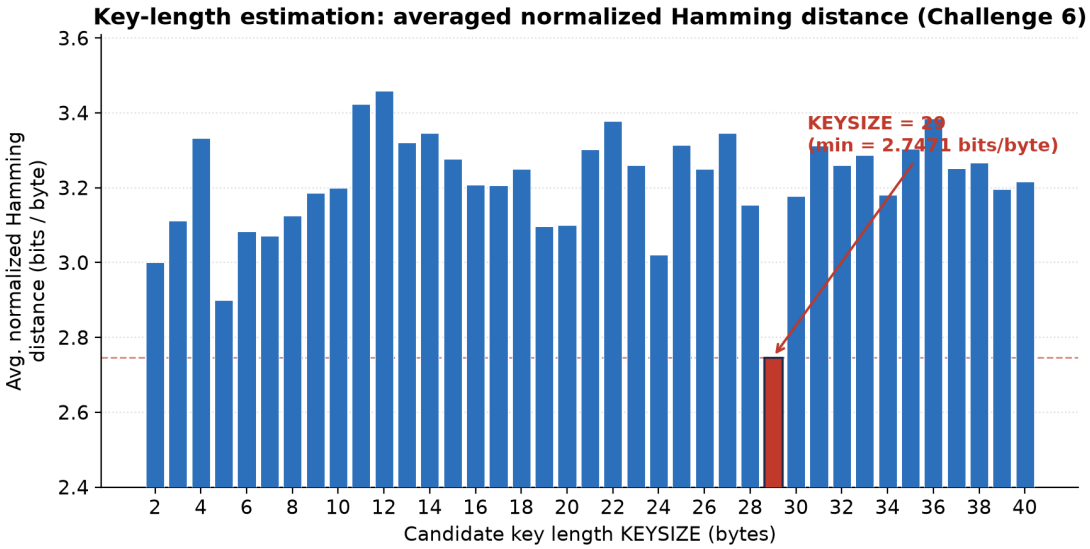

Figure 3. Averaged normalized Hamming distance $S(\text{KEYSIZE})$ of Equation (3) for every candidate key length 2–40. The true key length, 29, is the clear global minimum (2.7471 bits per byte), separated from all competitors.

score is

$$\chi^2 = \sum_{c \in \mathcal{A}} \frac{(O_c - E_c)^2}{E_c} + 100 \cdot (\#\text{ non-printable bytes}), \tag{4}$$

where $\mathcal{A}$ is the alphabet of scored characters. The additive penalty term rejects trial keys that produce control bytes or other non-text output (tab, newline and carriage return are exempted as legitimate whitespace): a correct key byte yields readable text and a low χ^2, whereas an incorrect one scatters the byte distribution and usually emits non-printable characters, inflating the score. The key byte minimizing Equation (4) is adopted for that column, and concatenating the m winners in column order reconstructs the full key.

To choose among candidate key lengths robustly, the top five ranked sizes from the Hamming phase are each fully solved and decrypted. Two independent quality measures are computed for each: the mean per-column χ^2 (how English-like the columns become) and the printable-byte ratio of the full decryption. The candidate that achieves a high printable ratio and the lowest χ^2 is selected. This cross-check guards against the well-known trap in which a divisor or small multiple of the true period scores acceptably on the Hamming test alone.

3.4 Results

The two phases agree decisively on KEYSIZE = 29 (Table 2). It is simultaneously the minimum-Hamming candidate (2.7471 bits/byte) and, by a wide margin, the minimum-χ^2 candidate: its mean per-column χ^2 of 40.93 is more than an order of magnitude below the next-best value (619.99 for KEYSIZE = 24) and roughly two orders below the short spurious candidates 5 and 2 (Figure 6). Its printable ratio is a perfect 1.0000. No other candidate yields coherent text.

The recovered artefacts are:

- **Key length:** 29 bytes.
- **Key (ASCII):** `Terminator X: Bring the noise`
- **Key (hex):** `5465726d696e61746f7220583a204272696e6720746865206e6f697365`
- **Plaintext:** the full 2,876-byte lyric to Vanilla Ice's "Play That Funky Music" (562 words, printable ratio 1.0000), reproduced in full in Appendix D.

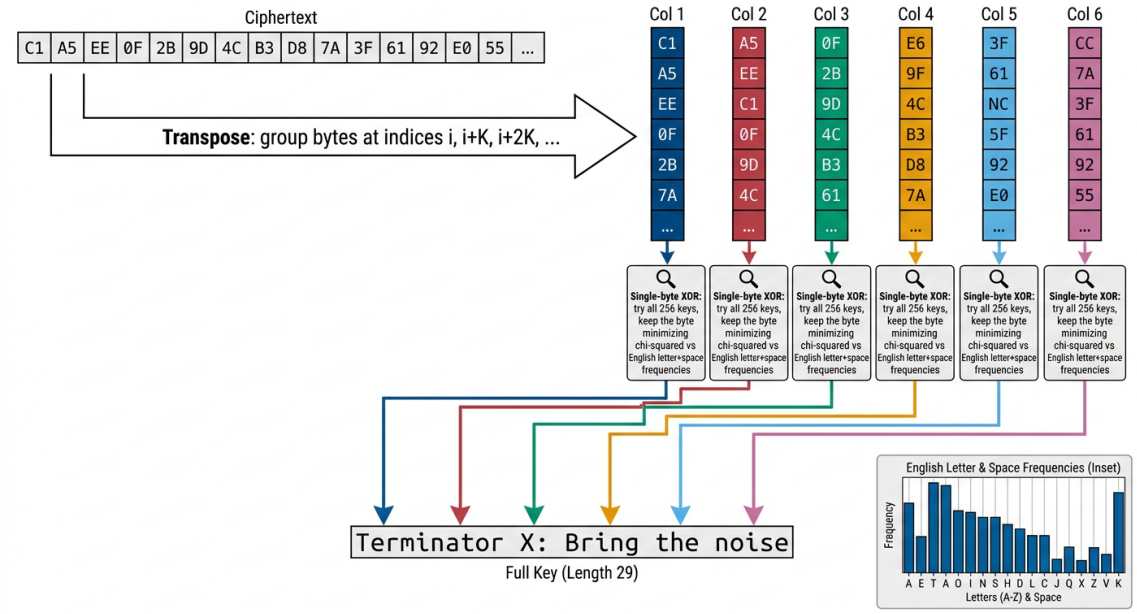

Figure 4. Transposition and column-wise solution. The ciphertext is regrouped into $m = 29$ columns; each column is a single-byte-XOR cipher solved by trying all 256 key bytes and keeping the one that minimizes the English χ^2 score of Equation (4). The per-column winners concatenate into the full key `Terminator X: Bring the noise`.

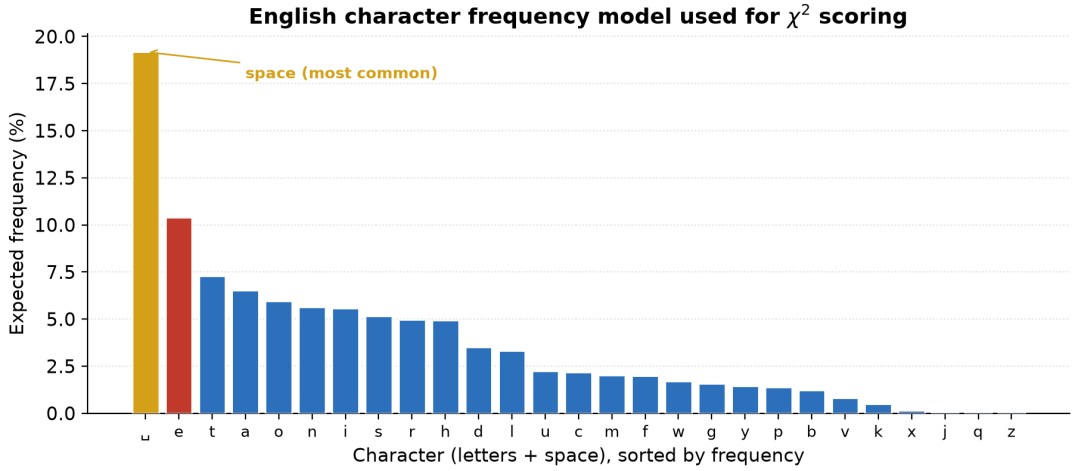

Figure 5. The English character-frequency model f_c used in Equation (4). The space is deliberately included as the most common character ($\approx 19\%$); *e*, *t*, *a*, *o* follow. This reference distribution is what distinguishes a correct single-byte key (text that matches these proportions) from the 255 incorrect ones.

Table 2. Top-five candidate key lengths, fully solved. KEYSIZE 29 minimizes both diagnostics and is the only candidate producing coherent English.

KEYSIZE	Avg. norm. Hamming	Mean column χ^2	Printable ratio
29	**2.7471**	**40.93**	**1.0000**
5	2.9000	3335.29	0.9524
2	3.0000	8618.17	0.9531
24	3.0208	619.99	0.9607
7	3.0714	2332.18	0.9565

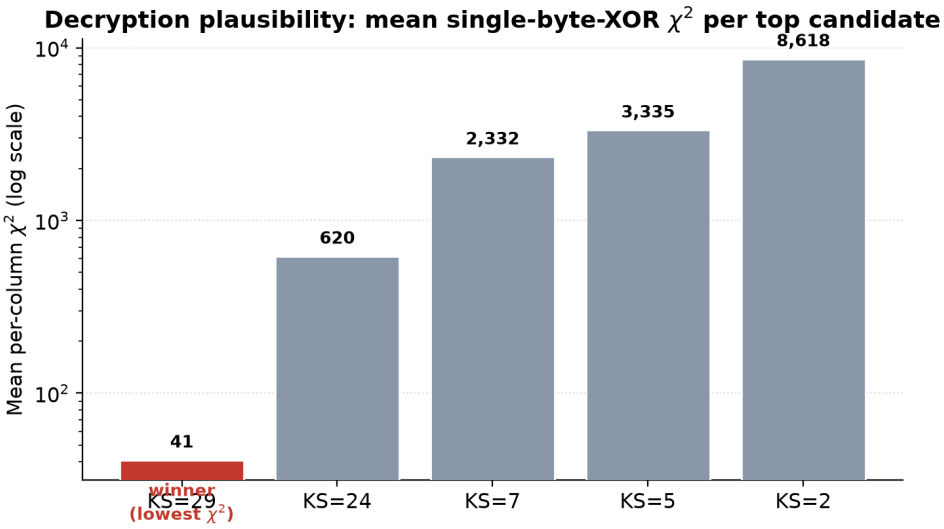

Figure 6. Mean per-column χ^2 (log scale) for the five candidate key lengths after full decryption. KEYSIZE 29 (crimson) is separated from all competitors by more than an order of magnitude, confirming the Hamming-based choice.

The decryption begins "*I'm back and I'm ringin' the bell / A rockin' on the mike while the fly girls yell ...*" and continues coherently to the end of the file, confirming that both the key and every one of the 29 column solutions are correct.

4 Challenge 8 — Detecting AES in ECB Mode

4.1 AES, block modes, and the determinism of ECB

AES is a 128-bit block cipher: under a fixed key K it defines a bijection E_K on the space of 16-byte blocks [9, 10]. A block cipher only enciphers one block; a *mode of operation* extends it to messages of arbitrary length [11]. The simplest mode, Electronic Codebook, partitions the padded plaintext into 16-byte blocks $P_1, P_2, \ldots, P_t$ and encrypts each one in isolation,

$$C_i = E_K(P_i), \qquad 1 \le i \le t. \tag{5}$$

The name is apt: ECB is literally a codebook lookup, and it inherits the codebook's fatal property. Because E_K is a deterministic function with no chaining, no initialization vector, and no counter, Equation (5) guarantees

$$P_i = P_j \implies C_i = C_j. \tag{6}$$

That is, *equal plaintext blocks always produce equal ciphertext blocks* (Figure 7). Chaining and stream-like modes—CBC, CFB, OFB, CTR—break this implication by mixing each block with a previous ciphertext block or with a per-block counter/nonce, so that repeated plaintext blocks map to *different* ciphertext blocks [11, 12]. The determinism of Equation (6) is not a flaw in AES; it is a flaw in the mode, and it is exactly the signal a detector can exploit.

4.2 Detection criterion

Real plaintexts—English text, tabular data, images, padded records—routinely contain repeated 16-byte segments. By Equation (6), any such repetition is copied verbatim into an ECB ciphertext as repeated 16-byte ciphertext blocks. A ciphertext produced by any semantically secure mode, by contrast, is computationally indistinguishable from random and therefore

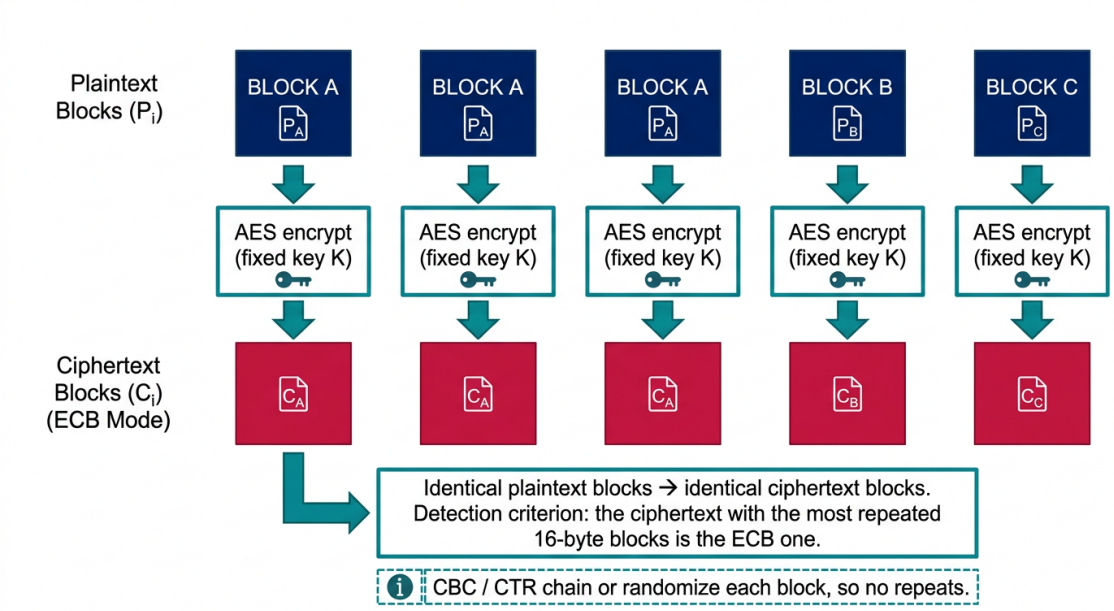

Figure 7. ECB determinism (Equations (5)–(6)). Each plaintext block is encrypted independently under the fixed key, so identical plaintext blocks (`BLOCK A`) yield identical ciphertext blocks. Randomized or chained modes (CBC, CTR) avoid this and leave no such repeats.

exhibits repeated blocks only with negligible probability (for 16-byte blocks the birthday bound makes an accidental collision astronomically unlikely across a mere ten blocks). The detection criterion follows immediately:

> *Hex-decode each ciphertext, partition it into* 16*-byte blocks, and count how many blocks are duplicates. The ECB-encrypted ciphertext is the line with the largest number of repeated blocks; equivalently, the line with the fewest distinct blocks relative to its total block count.*

Operationally, for a line decoding to blocks $b_1, \ldots, b_t$ we form the multiset of blocks, count occurrences, and report repeats $= t - |\{\text{distinct blocks}\}|$. Ties are broken toward the earliest line. This is a linear-time scan with no knowledge of the key and no decryption whatsoever.

4.3 Results

Scanning all 204 ciphertexts (each exactly $t = 10$ blocks) yields a stark, unambiguous verdict (Figure 8, top). **Exactly one** line shows any block repetition at all: line **133** (1-based), which contains 3 repeated blocks. Every other line has 10 distinct blocks and a repeat count of zero. Line 133 is therefore the AES-ECB ciphertext, matching the canonical Cryptopals answer.

The block map of line 133 (Figure 8, bottom) makes the fingerprint concrete: of its ten 16-byte blocks, one distinct block, `08649af70dc06f4fd5d2d69c744cd283`, recurs **four** times (in block positions 2, 4, 6 and 8), leaving 7 unique blocks out of 10 (Table 3). No decryption was performed or required; the detection rests solely on the combinatorial structure guaranteed by Equation (6).

5 Discussion

5.1 Why both schemes fail the modern standard

The two attacks, though superficially different, share a single root cause: *the encryption transformation is a deterministic function of the plaintext that preserves an exploitable invariant.*

Table 3. AES-ECB detection result for Challenge 8.

Quantity	Value
Detected line (1-based)	**133**
Total 16-byte blocks / line	10
Unique blocks on line 133	7
Repeated block count	3
Distinct repeated blocks	1
Most frequent block (hex)	`08649af70dc06f4fd5d2d69c744cd283` (×4)
Lines with any repetition	1 of 204 (unambiguous)

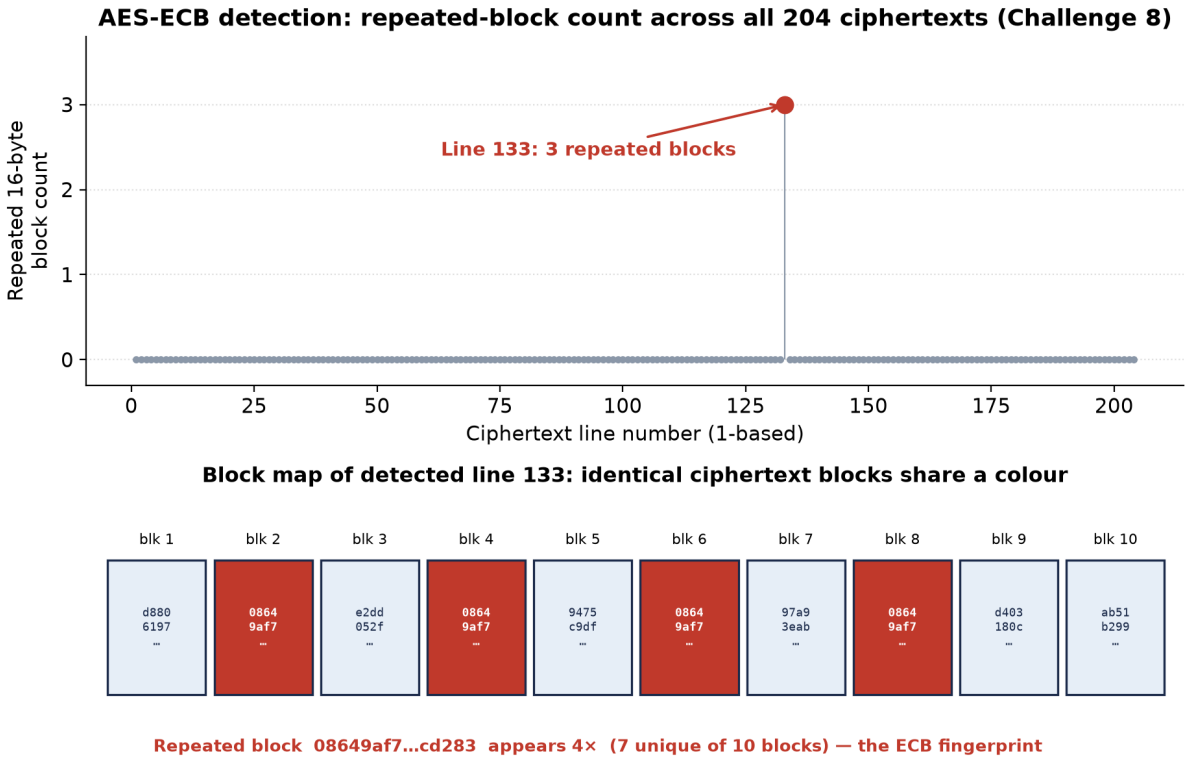

Figure 8. AES-ECB detection diagnostics. *Top:* the number of repeated 16-byte blocks for each of the 204 ciphertexts; only line 133 is non-zero. *Bottom:* the ten blocks of line 133, coloured so that identical blocks share a colour—the block `08649af7...cd283` appears four times, the telltale ECB fingerprint.

For repeating-key XOR the invariant is the per-residue-class letter distribution; for ECB it is block-level equality. The modern security definition—indistinguishability under chosen-plaintext attack (IND-CPA), the computational refinement of Shannon's perfect secrecy [1, 2, 13]—forbids exactly this. IND-CPA requires that an adversary who may even choose the plaintexts cannot distinguish the encryption of one chosen message from another. A deterministic scheme fails this immediately: encrypting the same message twice produces the same ciphertext, so an adversary distinguishes "same vs. different" with certainty. Repeating-key XOR fails because a short key forces the same key stream to recur; ECB fails at the finer granularity of the individual block. In both cases the fix is the same conceptual move that Goldwasser and Micali introduced to public-key encryption [2]: *randomize*, so that identical inputs almost never yield identical outputs. It is worth stressing that this randomization discipline is orthogonal to the symmetric-versus-asymmetric distinction: the public-key revolution launched by Diffie and Hellman [21] and realized by RSA [22] solved the problem of *distributing* keys, but a public-key

scheme deployed deterministically leaks in exactly the same way, which is why probabilistic encryption became the baseline expectation there too.

5.2 Repeating-key XOR and the fragility of the one-time pad

The XOR construction of Equation (1) is instructive precisely because it is one keystroke away from the one and only cipher with a proof of perfect secrecy. Vernam's cipher and Shannon's analysis show that a uniformly random pad, as long as the message and never reused, leaks nothing [8, 1]. Every departure from those conditions is catastrophic. A short, repeated key turns the pad into an interleaving of mono-alphabetic ciphers breakable by century-old methods [6, 7]; and even a full-length pad, if reused across two messages, yields $C_1 \oplus C_2 = P_1 \oplus P_2$, from which crib-dragging recovers both plaintexts. The practical lesson, emphasized throughout the modern literature [17, 23, 24], is that key management—length, randomness, and uniqueness—is not a detail around a cipher but the substance of its security. Contemporary stream ciphers such as the ChaCha20 construction standardized in RFC 8439 [16] keep the one-time-pad intuition but generate the keystream from a nonce and counter so that the pad is effectively message-length and never repeats under a given key.

5.3 ECB and the leakage of structure

The ECB result generalizes far beyond a puzzle. Because ECB copies plaintext equality into ciphertext equality (Equation (6)), it leaks the repetition structure of *any* data it protects: the contours of a bitmap image survive encryption, and record-structured data (fixed-format logs, database columns, password tables) reveal which entries are equal. This is not hypothetical folklore; deterministic encryption of a structured field was the mechanism behind real-world password-database exposures, and it is why the NIST guidance [11] and comparative mode evaluations [12] treat ECB as unsuitable for general confidentiality. The remedy is a randomized mode. CBC introduces a random initialization vector so that even identical messages encrypt differently—though CBC brings its own implementation hazards, most notably the padding-oracle attacks of Vaudenay [25], which, like the challenges here, exploit side information rather than the AES core. Counter (CTR) mode turns AES into a nonce-based stream cipher with no block-repetition signature at all [11].

5.4 From confidentiality to authenticated encryption

Neither randomization alone nor any of the classical modes provides integrity: an attacker who cannot read a ciphertext may still be able to tamper with it, and attacks such as padding oracles [25] and timing side channels [26] show that confidentiality-only constructions leak through their error and timing behaviour. The modern consensus is therefore authenticated encryption with associated data (AEAD), which binds confidentiality and integrity in a single primitive with a clean security definition [14]. Widely deployed AEAD schemes— AES-GCM [15] and ChaCha20-Poly1305 [16]—are randomized (nonce-based), leave no ECB-style block-repetition signature, and reject tampered ciphertexts outright. They are the direct descendants of the lessons these two challenges teach in miniature.

5.5 Methodological remarks and limitations

Both attacks are statistical, and their reliability scales with data. The Hamming key-length test sharpens as more block pairs are averaged and as the ciphertext lengthens; on very short messages the normalized distances of nearby key sizes can overlap, which is why the second-phase χ^2/printable cross-check (Section 3.3) is essential rather than optional. The English frequency model of Figure 5 assumes English-language, largely alphabetic plaintext; a different language or a binary payload would require a matched reference distribution, but the algorithmic skeleton is

unchanged. For ECB detection the criterion is exact rather than statistical whenever a repeated plaintext block exists, as here; its only blind spot is an ECB ciphertext whose plaintext happens to contain no repeated 16-byte block, in which case the mode leaks nothing on that particular message even though it remains insecure in general. None of these caveats affected the present datasets, where both results are unambiguous and exactly reproduce the canonical answers.

6 Conclusion

Two short, dependency-free programs recover, from public challenge data alone, a complete repeating-key XOR key and plaintext and pinpoint an AES-ECB ciphertext in a corpus of 204. For Challenge 6 the averaged normalized Hamming distance identifies a 29-byte key length, and per-column χ^2 frequency analysis recovers the key `Terminator X: Bring the noise` and the full 2,876-byte plaintext at printable ratio 1.0000. For Challenge 8 the deterministic, block-wise nature of ECB betrays line **133**, the only ciphertext of the 204 exhibiting repeated 16-byte blocks. Neither attack touches the strength of the underlying primitive; both read structure that a properly randomized and authenticated mode would have destroyed. The enduring lesson—from Shannon and Vernam to modern AEAD—is that confidentiality is a property of *how* ciphers are used, and that any deterministic residue of the plaintext is, in the hands of a patient analyst, the whole message.

References

[1] C. E. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[2] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[3] A. Kerckhoffs, "La cryptographie militaire," *Journal des sciences militaires*, vol. IX, pp. 5–38, 161–191, 1883.

[4] T. P. Rivest-Shamir, S. Devlin, A. Balducci, and M. Wielgoszewski, "The cryptopals crypto challenges — set 1." https://cryptopals.com/sets/1, 2018. Accessed 2026-07-12. Challenge 6 data: https://cryptopals.com/static/challenge-data/6.txt; Challenge 8 data: https://cryptopals.com/static/challenge-data/8.txt.

[5] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *Journal of Cryptology*, vol. 4, no. 1, pp. 3–72, 1991.

[6] F. W. Kasiski, *Die Geheimschriften und die Dechiffrir-Kunst*. Berlin: E. S. Mittler und Sohn, 1863.

[7] W. F. Friedman, "The index of coincidence and its applications in cryptography," Riverbank Publication 22, Riverbank Laboratories, Department of Ciphers, Geneva, Illinois, 1922.

[8] G. S. Vernam, "Cipher printing telegraph systems for secret wire and radio telegraphic communications," *Transactions of the American Institute of Electrical Engineers*, vol. 45, pp. 295–301, 1926.

[9] National Institute of Standards and Technology, "Advanced encryption standard (aes)," Federal Information Processing Standards Publication FIPS 197 (update 1), U.S. Department of Commerce, National Institute of Standards and Technology, 2023.

[10] J. Daemen and V. Rijmen, *The Design of Rijndael: AES — The Advanced Encryption Standard*. Information Security and Cryptography, Berlin, Heidelberg: Springer-Verlag, 2002.

[11] M. Dworkin, "Recommendation for block cipher modes of operation: Methods and techniques," NIST Special Publication 800-38A, National Institute of Standards and Technology, 2001.

[12] P. Rogaway, "Evaluation of some blockcipher modes of operation," tech. rep., Cryptography Research and Evaluation Committees (CRYPTREC), Government of Japan, 2011.

[13] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 394–403, IEEE Computer Society, 1997.

[14] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology — ASIACRYPT 2000*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 531–545, Springer, 2000.

[15] D. A. McGrew and J. Viega, "The security and performance of the galois/counter mode (gcm) of operation," in *Progress in Cryptology — INDOCRYPT 2004*, vol. 3348 of *Lecture Notes in Computer Science*, pp. 343–355, Springer, 2004.

[16] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," Request for Comments RFC 8439, Internet Engineering Task Force (IETF), 2018.

[17] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Boca Raton, FL: CRC Press, 3rd ed., 2021.

[18] D. R. Stinson and M. B. Paterson, *Cryptography: Theory and Practice*. Boca Raton, FL: CRC Press, 4th ed., 2018.

[19] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[20] K. Pearson, "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

[21] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[22] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[23] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering: Design Principles and Practical Applications*. Indianapolis, IN: Wiley, 2010.

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1996.

[25] S. Vaudenay, "Security flaws induced by CBC padding — applications to SSL, IPSEC, WTLS . . . ," in *Advances in Cryptology — EUROCRYPT 2002*, vol. 2332 of *Lecture Notes in Computer Science*, pp. 534–545, Springer, 2002.

[26] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology — CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113, Springer, 1996.

A Attack code — Challenge 6 (break repeating-key XOR)

```python
#!/usr/bin/env python3
"""Cryptopals Challenge 6: Break repeating-key XOR.

A zero-dependency (standard library only) implementation of a mathematically
rigorous pipeline that:
  1. Computes exact bit-level Hamming distances (popcount of XOR).
  2. Estimates the repeating-key length (KEYSIZE) via averaged normalized
     Hamming distance over multiple block pairs.
  3. Transposes the ciphertext into KEYSIZE single-byte-XOR streams.
  4. Solves each stream with chi-squared frequency analysis over all 256 keys.
  5. Reconstructs the full key and decrypts the ciphertext.

Usage:
    uv run python workflow/02_break_repeating_xor.py \
        --input data/challenge6.txt \
        --out results/challenge6_result.json
"""

from __future__ import annotations

import argparse
import base64
import json
import logging
from itertools import combinations
from pathlib import Path

# ---------------------------------------------------------------------------- #
# Logging
# ---------------------------------------------------------------------------- #
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    datefmt="%H:%M:%S",
)
log = logging.getLogger("break_repeating_xor")


# ---------------------------------------------------------------------------- #
# Step 1: Bit-level Hamming distance
# ---------------------------------------------------------------------------- #
def hamming_distance(a: bytes, b: bytes) -> int:
    """Exact bit-level Hamming distance between two equal-length byte arrays.

    The distance is the population count (number of set bits) of the bitwise
    XOR of the two inputs.

    Parameters
    ----------
    a, b : bytes
        Equal-length byte sequences.

    Returns
    -------
    int
        Number of differing bits.
    """
    if len(a) != len(b):
        raise ValueError(f"Inputs must be equal length: {len(a)} != {len(b)}")
    # int.bit_count() is exact popcount (Python 3.10+); fall back to bin().count
    return sum(
        (x ^ y).bit_count() if hasattr(int, "bit_count") else bin(x ^ y).count("1")
        for x, y in zip(a, b)
    )


# ---------------------------------------------------------------------------- #
# Step 2: Key-length (KEYSIZE) estimation
# ---------------------------------------------------------------------------- #
def score_keysize(data: bytes, keysize: int, n_blocks: int = 4) -> float:
```

```
    """Average normalized Hamming distance across all pairwise blocks.

    Takes the first ``n_blocks`` consecutive blocks of ``keysize`` bytes and
    averages the normalized (per-byte) Hamming distance over every unique pair
    of blocks. Lower scores indicate more likely key lengths.
    """
    max_blocks = len(data) // keysize
    blocks_to_use = min(n_blocks, max_blocks)
    if blocks_to_use < 2:
        return float("inf")

    blocks = [data[i * keysize:(i + 1) * keysize] for i in range(blocks_to_use)]
    distances = [
        hamming_distance(b1, b2) / keysize
        for b1, b2 in combinations(blocks, 2)
    ]
    return sum(distances) / len(distances)


def rank_keysizes(
    data: bytes, min_size: int = 2, max_size: int = 40, n_blocks: int = 4
) -> list[tuple[int, float]]:
    """Rank candidate key sizes by ascending averaged normalized Hamming score."""
    scored = []
    for ks in range(min_size, max_size + 1):
        if len(data) < ks * 2:
            continue
        scored.append((ks, score_keysize(data, ks, n_blocks)))
    scored.sort(key=lambda kv: kv[1])
    return scored


# ---------------------------------------------------------------------------- #
# Step 3: Transposition
# ---------------------------------------------------------------------------- #
def transpose_blocks(data: bytes, keysize: int) -> list[bytes]:
    """Split ciphertext into ``keysize`` streams.

    Stream ``i`` contains every ``keysize``-th byte starting at index ``i``
    (bytes at indices i, i+keysize, i+2*keysize, ...), i.e. all bytes that were
    XORed with the same key byte.
    """
    return [bytes(data[i::keysize]) for i in range(keysize)]


# ---------------------------------------------------------------------------- #
# Step 4: Single-byte XOR frequency analysis
# ---------------------------------------------------------------------------- #
# Expected relative frequencies of English characters (letters + space).
# Derived from standard English letter frequency tables, with space added as
# the most common character. Values are proportions that sum to ~1.0.
ENGLISH_FREQ: dict[str, float] = {
    "a": 0.0651738, "b": 0.0124248, "c": 0.0217339, "d": 0.0349835,
    "e": 0.1041442, "f": 0.0197881, "g": 0.0158610, "h": 0.0492888,
    "i": 0.0558094, "j": 0.0009033, "k": 0.0050529, "l": 0.0331490,
    "m": 0.0202124, "n": 0.0564513, "o": 0.0596302, "p": 0.0137645,
    "q": 0.0008606, "r": 0.0497563, "s": 0.0515760, "t": 0.0729357,
    "u": 0.0225134, "v": 0.0082903, "w": 0.0171272, "x": 0.0013692,
    "y": 0.0145984, "z": 0.0007836, " ": 0.1918182,
}


def chi_squared_score(text: bytes) -> float:
    """Chi-squared statistic of ``text`` against English frequencies.

    Lower values indicate a closer match to English. Non-printable control
    bytes (excluding tab/newline/carriage-return) incur a heavy penalty so that
    keys producing binary garbage are rejected.
    """
    n = len(text)
    if n == 0:
        return float("inf")

```

```
    counts: dict[str, int] = {c: 0 for c in ENGLISH_FREQ}
    penalty = 0
    for byte in text:
        ch = chr(byte)
        lower = ch.lower()
        if lower in counts:
            counts[lower] += 1
        elif byte in (9, 10, 13):  # tab, LF, CR are acceptable whitespace
            pass
        elif 32 <= byte < 127:  # other printable ASCII (digits, punctuation)
            pass
        else:  # non-printable / high bytes: strong signal of a wrong key
            penalty += 1

    chi2 = 0.0
    for ch, expected_prop in ENGLISH_FREQ.items():
        expected = expected_prop * n
        observed = counts[ch]
        chi2 += (observed - expected) ** 2 / expected
    return chi2 + penalty * 100.0


def solve_single_byte_xor(block: bytes) -> tuple[int, float]:
    """Return the (best_key_byte, best_chi2) minimizing the chi-squared score."""
    best_key = 0
    best_score = float("inf")
    for key in range(256):
        decrypted = bytes(b ^ key for b in block)
        score = chi_squared_score(decrypted)
        if score < best_score:
            best_score = score
            best_key = key
    return best_key, best_score


# ---------------------------------------------------------------------------- #
# Step 5: Key reconstruction & decryption
# ---------------------------------------------------------------------------- #
def repeating_key_xor(data: bytes, key: bytes) -> bytes:
    """XOR ``data`` with a repeating ``key``."""
    return bytes(b ^ key[i % len(key)] for i, b in enumerate(data))


def recover_key(data: bytes, keysize: int) -> bytes:
    """Recover the full repeating key for a given key size."""
    blocks = transpose_blocks(data, keysize)
    key_bytes = bytearray()
    for block in blocks:
        k, _ = solve_single_byte_xor(block)
        key_bytes.append(k)
    return bytes(key_bytes)


def printable_ratio(data: bytes) -> float:
    """Fraction of bytes that are printable ASCII or standard whitespace."""
    if not data:
        return 0.0
    good = sum(1 for b in data if 32 <= b < 127 or b in (9, 10, 13))
    return good / len(data)


# ---------------------------------------------------------------------------- #
# Main pipeline
# ---------------------------------------------------------------------------- #
def main() -> None:
    parser = argparse.ArgumentParser(description="Break repeating-key XOR (Cryptopals #6).")
    parser.add_argument("--input", required=True, help="Path to base64 ciphertext file.")
    parser.add_argument("--out", required=True, help="Path to output JSON result.")
    parser.add_argument("--min-keysize", type=int, default=2)
    parser.add_argument("--max-keysize", type=int, default=40)
    parser.add_argument("--n-blocks", type=int, default=4,
                        help="Number of consecutive blocks used per keysize estimate.")
    parser.add_argument("--top-k", type=int, default=5,
```

```
                        help="Number of top candidate key sizes to fully test.")
    args = parser.parse_args()

    in_path = Path(args.input)
    out_path = Path(args.out)
    out_path.parent.mkdir(parents=True, exist_ok=True)

    # --- Self-test the Hamming distance on the canonical Cryptopals example ---
    known_a = b"this is a test"
    known_b = b"wokka wokka!!!"
    hd = hamming_distance(known_a, known_b)
    assert hd == 37, f"Hamming self-test failed: expected 37, got {hd}"
    log.info("Hamming distance self-test passed (37 bits for canonical example).")

    # --- Load & base64-decode ciphertext ---
    raw = in_path.read_text().strip()
    ciphertext = base64.b64decode("".join(raw.split()), validate=True)
    log.info("Decoded ciphertext: %d bytes.", len(ciphertext))

    # --- Step 2: rank key sizes ---
    ranked = rank_keysizes(
        ciphertext, args.min_keysize, args.max_keysize, args.n_blocks
    )
    log.info("Top candidate key sizes (keysize -> avg normalized Hamming):")
    for ks, sc in ranked[:args.top_k]:
        log.info("  KEYSIZE=%2d  score=%.4f", ks, sc)

    # --- Steps 3-5: fully solve each of the top-k candidates ---
    candidates = []
    for ks, hscore in ranked[:args.top_k]:
        key = recover_key(ciphertext, ks)
        plaintext = repeating_key_xor(ciphertext, key)
        # Total decryption plausibility = mean chi-squared per transposed block.
        total_chi2 = sum(
            chi_squared_score(bytes(b ^ key[i] for b in transpose_blocks(ciphertext, ks)[i]))
            for i in range(ks)
        ) / ks
        pr = printable_ratio(plaintext)
        candidates.append({
            "keysize": ks,
            "hamming_score": hscore,
            "mean_block_chi2": total_chi2,
            "printable_ratio": pr,
            "key": key,
            "plaintext": plaintext,
        })
        log.info(
            "KEYSIZE=%2d  hamming=%.4f  mean_chi2=%.2f  printable=%.4f",
            ks, hscore, total_chi2, pr,
        )

    # Select the best candidate: highest printable ratio, then lowest chi2.
    best = min(
        candidates,
        key=lambda c: (-(c["printable_ratio"] >= 0.95), c["mean_block_chi2"]),
    )

    key_bytes = best["key"]
    plaintext_bytes = best["plaintext"]
    try:
        key_ascii = key_bytes.decode("ascii")
    except UnicodeDecodeError:
        key_ascii = key_bytes.decode("latin-1")
    plaintext_str = plaintext_bytes.decode("utf-8", errors="replace")

    log.info("=" * 60)
    log.info("RECOVERED KEY LENGTH : %d", len(key_bytes))
    log.info("RECOVERED KEY (ASCII): %r", key_ascii)
    log.info("PLAINTEXT PREVIEW    : %r", plaintext_str[:80])
    log.info("PRINTABLE RATIO      : %.4f", best["printable_ratio"])
    log.info("=" * 60)

    result = {
```

```
        "challenge": "Cryptopals Set 1, Challenge 6: Break repeating-key XOR",
        "input_file": str(in_path),
        "ciphertext_bytes": len(ciphertext),
        "hamming_self_test_bits": hd,
        "keysize_search_range": [args.min_keysize, args.max_keysize],
        "n_blocks_per_estimate": args.n_blocks,
        "top_candidate_keysizes": [
            {
                "keysize": c["keysize"],
                "avg_normalized_hamming": round(c["hamming_score"], 6),
                "mean_block_chi2": round(c["mean_block_chi2"], 4),
                "printable_ratio": round(c["printable_ratio"], 6),
            }
            for c in candidates
        ],
        "recovered_key_length": len(key_bytes),
        "recovered_key_ascii": key_ascii,
        "recovered_key_hex": key_bytes.hex(),
        "printable_ratio": round(best["printable_ratio"], 6),
        "decrypted_plaintext": plaintext_str,
    }

    out_path.write_text(json.dumps(result, indent=2, ensure_ascii=False))
    log.info("Wrote structured result -> %s", out_path)


if __name__ == "__main__":
    main()
```

B Attack code — Challenge 8 (detect AES-ECB)

```python
#!/usr/bin/env python3
"""Cryptopals Challenge 8: Detect AES in ECB mode.

A zero-dependency (standard library only) detector that identifies which of a
set of hex-encoded ciphertexts was encrypted with AES in ECB mode.

Detection principle
--------------------
AES-ECB encrypts each 16-byte plaintext block independently and
deterministically: identical plaintext blocks map to identical ciphertext
blocks under a fixed key. Real-world plaintext (English text, structured data)
contains repeated 16-byte segments, so an ECB ciphertext betrays itself through
*repeated 16-byte ciphertext blocks*. CBC/CTR/etc. do not exhibit this because
each block depends on prior blocks / a counter.

The detection criterion is therefore: the line with the largest number of
repeated 16-byte blocks (equivalently, the smallest count of distinct blocks
relative to its total block count) is the ECB-encrypted line.

Usage:
    uv run python workflow/03_detect_aes_ecb.py \
        --input data/challenge8.txt \
        --out results/challenge8_result.json
"""

from __future__ import annotations

import argparse
import binascii
import json
import logging
from collections import Counter
from pathlib import Path

# ---------------------------------------------------------------------------- #
# Logging
# ---------------------------------------------------------------------------- #
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)-7s | %(message)s",
    datefmt="%H:%M:%S",
)
log = logging.getLogger("challenge8")

BLOCK_SIZE = 16  # AES block size in bytes (128 bits)


# ---------------------------------------------------------------------------- #
# Core analysis
# ---------------------------------------------------------------------------- #
def block_diagnostics(raw: bytes, block_size: int = BLOCK_SIZE) -> dict:
    """Compute ECB-relevant block statistics for one ciphertext.

    Parameters
    ----------
    raw : bytes
        The decoded ciphertext bytes.
    block_size : int
        Cipher block size (16 for AES).

    Returns
    -------
    dict
        Diagnostics including total/unique/repeated block counts and the
        repeated blocks with their frequencies.
    """
    n_full = len(raw) // block_size
    blocks = [raw[i * block_size:(i + 1) * block_size] for i in range(n_full)]

    counts = Counter(blocks)
```

```python
    n_unique = len(counts)
    n_repeated_blocks = n_full - n_unique  # duplicate occurrences beyond first

    # Blocks that appear more than once, hex-encoded with their frequency.
    repeats = {
        binascii.hexlify(block).decode(): freq
        for block, freq in counts.items()
        if freq > 1
    }

    trailing = len(raw) - n_full * block_size  # non-block-aligned remainder

    return {
        "total_bytes": len(raw),
        "block_size": block_size,
        "total_blocks": n_full,
        "trailing_bytes": trailing,
        "unique_blocks": n_unique,
        "repeated_block_count": n_repeated_blocks,
        "distinct_repeated_blocks": len(repeats),
        "repeated_blocks": repeats,
    }


def detect_ecb(lines: list[str]) -> dict:
    """Scan all ciphertext lines and identify the ECB-encrypted one.

    Parameters
    ----------
    lines : list[str]
        Hex-encoded ciphertext strings (one per input line).

    Returns
    -------
    dict
        Full detection result including the winning line and per-line summary.
    """
    per_line = []
    for idx, hexstr in enumerate(lines, start=1):
        raw = binascii.unhexlify(hexstr)
        diag = block_diagnostics(raw)
        diag["line_number"] = idx  # 1-based
        diag["ciphertext_hex"] = hexstr
        per_line.append(diag)

        if idx % 50 == 0:
            log.info("Scanned %d/%d ciphertext lines...", idx, len(lines))

    log.info("Scanned all %d ciphertext lines.", len(lines))

    # The ECB line maximizes repeated blocks. Tie-break deterministically on
    # the earliest line number (stable sort preserves input order on ties).
    winner = max(per_line, key=lambda d: d["repeated_block_count"])

    # Lines that show ANY block repetition (should normally be just the ECB one).
    candidates = [
        {
            "line_number": d["line_number"],
            "repeated_block_count": d["repeated_block_count"],
            "distinct_repeated_blocks": d["distinct_repeated_blocks"],
        }
        for d in per_line
        if d["repeated_block_count"] > 0
    ]
    candidates.sort(key=lambda d: (-d["repeated_block_count"], d["line_number"]))

    return {
        "winner": winner,
        "candidates_with_repeats": candidates,
        "n_lines": len(lines),
    }


```

```python
# ---------------------------------------------------------------------------- #
# I/O helpers
# ---------------------------------------------------------------------------- #
def read_hex_lines(path: Path) -> list[str]:
    """Read and validate newline-separated hex ciphertexts."""
    text = path.read_text(encoding="ascii")
    lines = [ln.strip() for ln in text.splitlines() if ln.strip()]
    if not lines:
        raise ValueError(f"No non-empty lines found in {path}")

    for i, ln in enumerate(lines, start=1):
        if len(ln) % 2 != 0:
            raise ValueError(f"Line {i}: odd-length hex string ({len(ln)} chars)")
        try:
            binascii.unhexlify(ln)
        except (binascii.Error, ValueError) as exc:
            raise ValueError(f"Line {i}: invalid hex ({exc})") from exc

    log.info(
        "Loaded %d hex lines from %s (each %d hex chars = %d bytes = %d blocks).",
        len(lines),
        path.name,
        len(lines[0]),
        len(lines[0]) // 2,
        (len(lines[0]) // 2) // BLOCK_SIZE,
    )
    return lines


def main() -> None:
    parser = argparse.ArgumentParser(description="Detect AES-ECB ciphertext (Cryptopals #8).")
    parser.add_argument(
        "--input",
        default="data/challenge8.txt",
        help="Path to newline-separated hex ciphertexts.",
    )
    parser.add_argument(
        "--out",
        default="results/challenge8_result.json",
        help="Path to write the JSON result.",
    )
    args = parser.parse_args()

    in_path = Path(args.input).resolve()
    out_path = Path(args.out).resolve()
    out_path.parent.mkdir(parents=True, exist_ok=True)

    log.info("=== Cryptopals Challenge 8: Detect AES-ECB ===")
    lines = read_hex_lines(in_path)
    result = detect_ecb(lines)

    winner = result["winner"]
    log.info("-" * 60)
    log.info("DETECTED ECB LINE: #%d (1-based)", winner["line_number"])
    log.info("  Total 16-byte blocks : %d", winner["total_blocks"])
    log.info("  Unique blocks        : %d", winner["unique_blocks"])
    log.info("  Repeated blocks      : %d", winner["repeated_block_count"])
    log.info("  Distinct repeats     : %d", winner["distinct_repeated_blocks"])
    for blk_hex, freq in winner["repeated_blocks"].items():
        log.info("    block %s x%d", blk_hex, freq)
    log.info("-" * 60)

    if len(result["candidates_with_repeats"]) == 1:
        log.info("Exactly one line shows block repetition -> unambiguous ECB detection.")
    else:
        log.info(
            "%d lines show block repetition; the maximum-repeat line is selected.",
            len(result["candidates_with_repeats"]),
        )

    # Assemble the persisted result.
    output = {
        "challenge": "Cryptopals Set 1, Challenge 8: Detect AES in ECB mode",
```

```
        "input_file": str(in_path),
        "n_ciphertexts": result["n_lines"],
        "block_size_bytes": BLOCK_SIZE,
        "detection_criterion": (
            "AES-ECB is deterministic and stateless per block: identical 16-byte "
            "plaintext blocks encrypt to identical 16-byte ciphertext blocks under "
            "a fixed key. The ECB-encrypted ciphertext is therefore the line with "
            "the largest number of repeated 16-byte blocks."
        ),
        "detected_line_number": winner["line_number"],  # 1-based
        "detected_line_index_0based": winner["line_number"] - 1,
        "repeated_block_count": winner["repeated_block_count"],
        "unique_blocks": winner["unique_blocks"],
        "total_blocks": winner["total_blocks"],
        "distinct_repeated_blocks": winner["distinct_repeated_blocks"],
        "repeated_blocks_hex_with_frequency": winner["repeated_blocks"],
        "detected_ciphertext_hex": winner["ciphertext_hex"],
        "all_lines_with_repeats": result["candidates_with_repeats"],
    }

    out_path.write_text(json.dumps(output, indent=2), encoding="ascii")
    log.info("Wrote result -> %s", out_path)
    log.info("=== Done ===")


if __name__ == "__main__":
    main()
```

C Program output

Challenge 6 (`challenge6_result.json`, abridged)

```
recovered_key_length : 29
recovered_key_ascii  : Terminator X: Bring the noise
recovered_key_hex    : 5465726d696e61746f7220583a204272696e6720746865206e6f697365
printable_ratio      : 1.0000
top KEYSIZE candidates (keysize, avg_norm_hamming, mean_chi2, printable):
   29   2.7471    40.93   1.0000   <-- winner
    5   2.9000  3335.29   0.9524
    2   3.0000  8618.17   0.9531
   24   3.0208   619.99   0.9607
    7   3.0714  2332.18   0.9565
hamming_self_test_bits : 37  (canonical "this is a test" vs "wokka wokka!!!")
```

Challenge 8 (`challenge8_result.json`, abridged)

```
detected_line_number   : 133   (1-based)
total_blocks           : 10
unique_blocks          : 7
repeated_block_count   : 3
distinct_repeated_blocks : 1
repeated block (hex)   : 08649af70dc06f4fd5d2d69c744cd283  (frequency 4)
lines_with_any_repeat  : 1 of 204  (unambiguous)
```

D Full recovered plaintext (Challenge 6)

```
I’m back and I’m ringin’ the bell
A rockin’ on the mike while the fly girls yell
In ecstasy in the back of me
Well that’s my DJ Deshay cuttin’ all them Z’s
Hittin’ hard and the girlies goin’ crazy
Vanilla’s on the mike, man I’m not lazy.

I’m lettin’ my drug kick in
It controls my mouth and I begin
To just let it flow, let my concepts go
My posse’s to the side yellin’, Go Vanilla Go!

Smooth ’cause that’s the way I will be
And if you don’t give a damn, then
Why you starin’ at me
So get off ’cause I control the stage
There’s no dissin’ allowed
I’m in my own phase
The girlies sa y they love me and that is ok
And I can dance better than any kid n’ play

Stage 2 -- Yea the one ya’ wanna listen to
It’s off my head so let the beat play through
So I can funk it up and make it sound good
1-2-3 Yo -- Knock on some wood
For good luck, I like my rhymes atrocious
Supercalafragilisticexpialidocious
I’m an effect and that you can bet
I can take a fly girl and make her wet.

I’m like Samson -- Samson to Delilah
There’s no denyin’, You can try to hang
But you’ll keep tryin’ to get my style
Over and over, practice makes perfect
But not if you’re a loafer.

You’ll get nowhere, no place, no time, no girls
Soon -- Oh my God, homebody, you probably eat
Spaghetti with a spoon! Come on and say it!

VIP. Vanilla Ice yep, yep, I’m comin’ hard like a rhino
Intoxicating so you stagger like a wino
So punks stop trying and girl stop cryin’
Vanilla Ice is sellin’ and you people are buyin’
’Cause why the freaks are jockin’ like Crazy Glue
Movin’ and groovin’ trying to sing along
All through the ghetto groovin’ this here song
Now you’re amazed by the VIP posse.

Steppin’ so hard like a German Nazi
Startled by the bases hittin’ ground
There’s no trippin’ on mine, I’m just gettin’ down
Sparkamatic, I’m hangin’ tight like a fanatic
You trapped me once and I thought that
You might have it
So step down and lend me your ear
```

```
’89 in my time! You, ’90 is my year.

You’re weakenin’ fast, YO! and I can tell it
Your body’s gettin’ hot, so, so I can smell it
So don’t be mad and don’t be sad
’Cause the lyrics belong to ICE, You can call me Dad
You’re pitchin’ a fit, so step back and endure
Let the witch doctor, Ice, do the dance to cure
So come up close and don’t be square
You wanna battle me -- Anytime, anywhere

You thought that I was weak, Boy, you’re dead wrong
So come on, everybody and sing this song

Say -- Play that funky music Say, go white boy, go white boy go
play that funky music Go white boy, go white boy, go
Lay down and boogie and play that funky music till you die.

Play that funky music Come on, Come on, let me hear
Play that funky music white boy you say it, say it
Play that funky music A little louder now
Play that funky music, white boy Come on, Come on, Come on
Play that funky music
```