Activity - Fundamental of Cryptography

Created by: Krerk Piromsopa, Ph.D

Overviews

In this activity, we will learn the basics of encryption. There are 3 exercises in this activity. Each exercise is designed to let you learn the concepts of cryptography. We will need:

- Imagemagick
- OpenSSL
- One of your favorite programming languages. You are welcome to do this exercise with any programming language. If you have no preference, use python.

Exercises

Q1.

(Cryptanalysis) Though encryption is primarily designed to preserve confidentiality and integrity of data, the mechanism itself is vulnerable to brute force (statistical analysis). In other words, the more we see the encrypted data, the easier we can hack it. In this exercise, you are asked to crack the following cipher text. Please provide the decrypted result and explain your strategy in decrypting this text.

Cipher text: "**PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE**".

a. Count the frequency of letters. List the top three most frequent characters.

```python
def count_freq(txt: str):
    dct = {}
    for ch in txt:
        if ch in dct:
            dct[ch] += 1
        else:
            dct[ch] = 1
    return dct


cipher = "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."
frq_dict = count_freq(cipher)

for k, v in sorted(frq_dict.items(), key=lambda x: x[1], reverse=True):
    print(k, v)
```

```
(base)
s/compu
  12
P 7
R 6
O 6
F 6
Q 5
L 4
S 3
A 3
Z 3
K 3
C 2
D 2
J 2
. 2
X 1
I 1
U 1
B 1
M 1
T 1
E 1
(base)
```

Answer: Top three "P" (7) then a three-way tie "R (6), O (6), F (6)". (whitespace excluded)

b. Knowing that this is English, what are commonly used three-letter words and two-letter words. Does the knowledge give you a hint on cracking the given text?

Answer:

Very common 2-letter words: is, of, to, in, on, at, an, as, by, or
Very common 3-letter words: the, and, for, not, but, you, are, his, her, was, one, our, out, day, get, has, him, how, new, now, old, see, two, way, who

Noticing repeated FP strongly suggests "is".
Seeing QDR in a spot where the often appears suggests QDR → "the".

Those two guesses give letter mappings:
F→i, P→s
Q→t, D→h, R→e

Now look at AFOPQ with those letters in place: A i O s t → "first" if A→f and O→r. This snowballs quickly.

c. Cracking the given text. Measure the time that you have taken to crack this message.

Answer:
Continuing the substitutions (from frequency + word shapes) yields:
Plaintext:
"**SECURITY IS THE FIRST CAUSE OF MISFORTUNE. THIS IS AN OLD GERMAN PROVERB.**"

Key partial mapping used (cipher → plain):
P→s, F→i, Q→t, D→h, R→e, A→f, O→r, C→c, Z→a, S→u, L→o, J→m, K→n, X→y, B→g, M→p, T→v, E→b, I→l, U→d
Time taken (human, frequency + pattern matching): ~ 30 minutes.

d. Create a simple python program for cracking the Caesar cipher text using brute force attack. Explain the design and demonstrate your software. (You may use an English dictionary for validating results.)

Answer:

```
import string
from collections import Counter

ALPH = string.ascii_uppercase


def caesar_decrypt(text: str, shift: int) -> str:
    """Shift letters backward by 'shift'. Non-letters pass through."""
    out = []
    for ch in text:
        up = ch.upper()
        if up in ALPH:
            idx = (ALPH.index(up) - shift) % 26
            dec = ALPH[idx]
            out.append(dec if ch.isupper() else dec.lower())
        else:
            out.append(ch)
    return "".join(out)


# English letter frequencies (percent) for chi-square scoring
EN_FREQ = {
    'A': 8.167,
    'B': 1.492,
    'C': 2.782,
    'D': 4.253,
    'E': 12.702,
    'F': 2.228,
    'G': 2.015,
    'H': 6.094,
    'I': 6.966,
    'J': 0.153,
    'K': 0.772,
    'L': 4.025,
    'M': 2.406,
    'N': 6.749,
    'O': 7.507,
    'P': 1.929,
    'Q': 0.095,
    'R': 5.987,
    'S': 6.327,
    'T': 9.056,
    'U': 2.758,
    'V': 0.978,
    'W': 2.360,
    'X': 0.150,
    'Y': 1.974,
    'Z': 0.074,
}
EN_TOTAL = sum(EN_FREQ.values())

COMMON_WORDS = {
    "THE",
    "AND",
    "OF",
    "TO",
    "IN",
    "IS",
    "IT",
    "YOU",
    "THAT",
    "FOR",
    "ON",
    "WITH",
    "AS",
    "ARE",
    "AT",
    "BE",
    "BY",
    "THIS",
    "I",
    "NOT",
    "OR",
    "FROM",
    "HAVE",
    "AN",
}


def chi_square_english(text: str) -> float:
    """Lower = closer to English distribution."""
    letters = [ch for ch in text.upper() if ch in ALPH]
    N = len(letters)
    if N == 0:
        return float("inf")
    obs = Counter(letters)
    chi = 0.0
    for L in ALPH:
        expected = EN_FREQ[L] / EN_TOTAL * N
        diff = obs.get(L, 0) - expected
        chi += diff * diff / (expected if expected > 0 else 1e-9)
    return chi


def english_word_score(text: str) -> int:
    """Very light dictionary check using common words."""
    tokens = "".join(ch if ch.isalpha() else " " for ch in text.upper()).split()
    return sum(tok in COMMON_WORDS for tok in tokens)


def brute_force_caesar(ciphertext: str, use_scoring: bool = True, top_k: int = 5):
    """Return top_k candidates (score, shift, plaintext)."""
    candidates = []
    for shift in range(26):
        pt = caesar_decrypt(ciphertext, shift)
        if use_scoring:
            chi = chi_square_english(pt)
            hits = english_word_score(pt)
            score = (-hits, chi)  # more hits is better; lower chi is better
        else:
            score = (0, 0)
        candidates.append((score, shift, pt))
    candidates.sort(key=lambda x: (x[0], x[1]))
    return candidates[:top_k], candidates


def main():
    # --- Demo ciphertext (Caesar shift 3) ---
    demo = "WKLV LV D FDHVDU FLSKHU."

    # Try all shifts and rank by (word hits desc, chi-square asc)
    top, all_candidates = brute_force_caesar(demo, use_scoring=True, top_k=5)

    # Best guess
    best_score, best_shift, best_plain = top[0]
    print("=== Best guess ===")
    print(f"Shift: {best_shift}")
    print(f"Score tuple (word_hits↑, chi-square↓) -> {best_score}")
    print(best_plain)

    # Top-5 preview
    print("\n=== Top 5 candidates ===")
    for score, shift, pt in top:
        hits = -score[0]  # we negated hits in the score
        chi = score[1]
        print(f"[shift {shift:2d}] hits={hits:2d} chi={chi:8.2f} | {pt}")

    # All 26 shifts (quick view)
    print("\n=== All 26 shifts ===")
    for _, shift, pt in sorted(all_candidates, key=lambda x: x[1]):
        print(f"[{shift:2d}] {pt}")


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

```
(base) pupipatsingkhorn@Pupipats-MacBook-Air computer-security %
=== Best guess ===
Shift: 3
Score tuple (word_hits↑, chi-square↓) -> (-2, 20.59176770300092)
THIS IS A CAESAR CIPHER.

=== Top 5 candidates ===
[shift  3] hits= 2 chi=   20.59 | THIS IS A CAESAR CIPHER.
[shift 21] hits= 1 chi=  862.25 | BPQA QA I KIMAIZ KQXPMZ.
[shift 18] hits= 0 chi=   35.12 | ESTD TD L NLPDLC NTASPC.
[shift  7] hits= 0 chi=   36.46 | PDEO EO W YWAOWN YELDAN.
[shift 16] hits= 0 chi=   85.94 | GUVF VF N PNRFNE PVCURE.

=== All 26 shifts ===
[ 0] WKLV LV D FDHVDU FLSKHU.
[ 1] VJKU KU C ECGUCT EKRJGT.
[ 2] UIJT JT B DBFTBS DJQIFS.
[ 3] THIS IS A CAESAR CIPHER.
[ 4] SGHR HR Z BZDRZQ BHOGDQ.
[ 5] RFGQ GQ Y AYCQYP AGNFCP.
[ 6] QEFP FP X ZXBPXO ZFMEBO.
[ 7] PDEO EO W YWAOWN YELDAN.
[ 8] OCDN DN V XVZNVM XDKCZM.
[ 9] NBCM CM U WUYMUL WCJBYL.
[10] MABL BL T VTXLTK VBIAXK.
[11] LZAK AK S USWKSJ UAHZWJ.
[12] KYZJ ZJ R TRVJRI TZGYVI.
[13] JXYI YI Q SQUIQH SYFXUH.
[14] IWXH XH P RPTHPG RXEWTG.
[15] HVWG WG O QOSGOF QWDVSF.
[16] GUVF VF N PNRFNE PVCURE.
[17] FTUE UE M OMQEMD OUBTQD.
[18] ESTD TD L NLPDLC NTASPC.
[19] DRSC SC K MKOCKB MSZROB.
[20] CQRB RB J LJNBJA LRYQNA.
[21] BPQA QA I KIMAIZ KQXPMZ.
[22] AOPZ PZ H JHLZHY JPWOLY.
[23] ZNOY OY G IGKYGX IOVNKX.
[24] YMNX NX F HFJXFW HNUMJW.
[25] XLMW MW E GEIWEV GMTLIV.
(base) pupipatsingkhorn@Pupipats-MacBook-Air computer-security %
```

1. Core Idea

A Caesar cipher shifts each letter of the alphabet by the same number of positions (0–25). To break it, we try all 26 possible shifts and check which result looks like real English.

2. Program Components

a) caesar_decrypt(text, shift)

- Takes the ciphertext and a shift value.
- For each character:
- If it's a letter: shift it backward by shift (wrap around A↔Z).
- If not a letter: leave it unchanged (spaces, punctuation).

- Returns the decrypted plaintext candidate.

b) English Scoring Functions
Since brute force generates 26 candidates, we need a way to guess which one is correct.
- Chi-square test (chi_square_english)
- Compares the frequency of letters in the candidate text with known English letter frequencies (E ≈ 12.7%, T ≈ 9.0%, etc.).
- Lower score = closer to natural English.
- Word hits (english_word_score)
- Splits text into tokens.
- Counts how many common English words (THE, AND, IS, etc.) appear.
- More hits = more likely to be real English.

c) brute_force_caesar(ciphertext, use_scoring=True, top_k=5)
- Loops over all 26 possible shifts.
- For each shift:
    1. Decrypt with caesar_decrypt.
    2. If scoring is enabled:
- Compute chi-square and word hits.
- Store a score tuple (-hits, chi). Negative hits so that more hits rank higher.
- Sorts all candidates by score.
- Returns the top k candidates and the full list.

d) main()
- Provides a demo ciphertext.
- Calls the brute force function.
- Prints:
- The best guess (top-scoring result).
- The top 5 candidates with their scores.
- All 26 shift results for manual inspection.

3. Why This Design Works
- Exhaustive search is possible because the Caesar keyspace is tiny (26 options).
- Statistical analysis (chi-square) makes the program automatically prioritize results that "look" like English text.
- Dictionary hits reinforce the result by detecting common short words, which usually appear in real sentences.
- Combining the two ensures high accuracy without needing a full dictionary.

4. Demonstration Example
Input ciphertext: WKLV LV D FDHVDU FLSKHU.
Output (best guess):
Shift: 3
THIS IS A CAESAR CIPHER.

Q2.

(Cryptanalysis on Symmetric Encryption) Vigenère is a complex version of the Caesar cipher. It is a polyalphabetic substitution.

a. Please review Kasiski examination 1 Vigenère.

Answer:

Ref. https://en.wikipedia.org/wiki/Kasiski_examination

Kasiski examination works by spotting repeated sequences in the ciphertext, measuring the distances between them, and using their greatest common divisors to estimate the key length. Once the key length is known, the Vigenère cipher can be split into several Caesar ciphers, each of which can be solved with simple frequency analysis.¶

Q3.

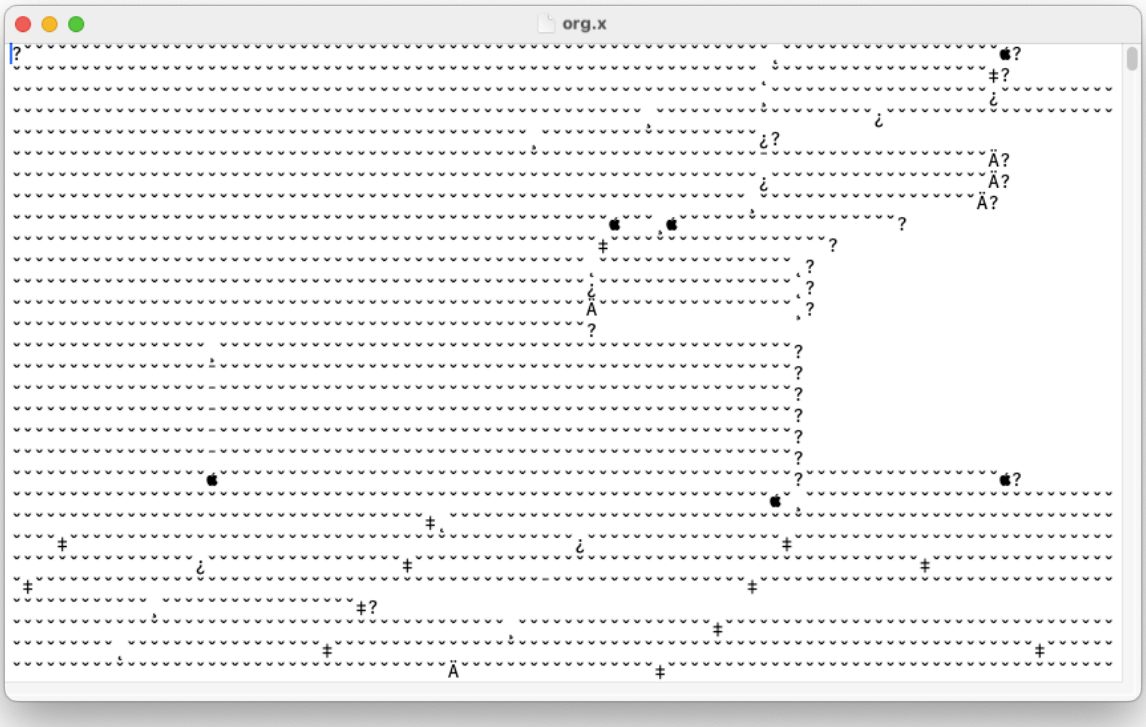
(Mode in Block Cipher) Block Cipher is designed to have more randomness in a block. However, an individual block still utilizes the same key. Thus, it is recommended to use a cipher mode with an initial vector, chaining or feedback between blocks. This exercise will show you the weakness of Electronic Code Book mode which does not include any initial vector, chaining or feedback.

a. Find a bitmap image that is larger than 2000x2000 pixels. Note that you may resize any image. To simplify the pattern, we will change it to bitmap (1-bit per pixel) using the portable bitmap format (pbm). In this example, we will use imagemagick for the conversion.
$ magick convert image.jpg -resize 2000x2000 org.pbm

Answer:

b. The NetPBM2 format is a naive image format.The first two lines contain a header (format and size in pixel). Depending on the format, the pixels can be represented in either binary and ascii. For our exercise, we prefer binary. However, we first have to take out the header to prevent the encryption from encoding the header. To do so, use your text editor (eg. vi, notepad) to take out the first two lines.
$ cp org.pbm org.x
$ vi org.x
P4
2000 2000
KR)B@HD@H

Answer:



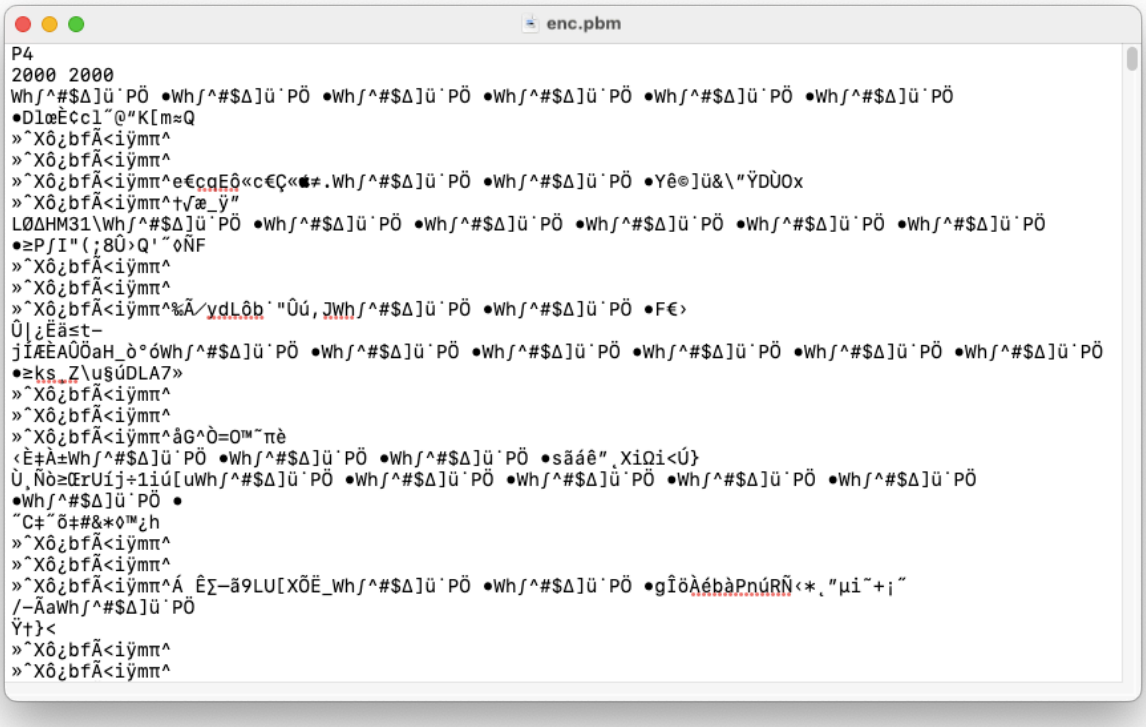
c. Encrypt the file with OpenSSL3 with any block cipher algorithm in ECB mode (no padding and no salt).
$ openssl enc -aes-256-ecb -in org.x -nosalt \ -out enc.x

Answer:

```
openssl enc -aes-256-ecb -in org.x -out enc.x -nosalt -nopad
```

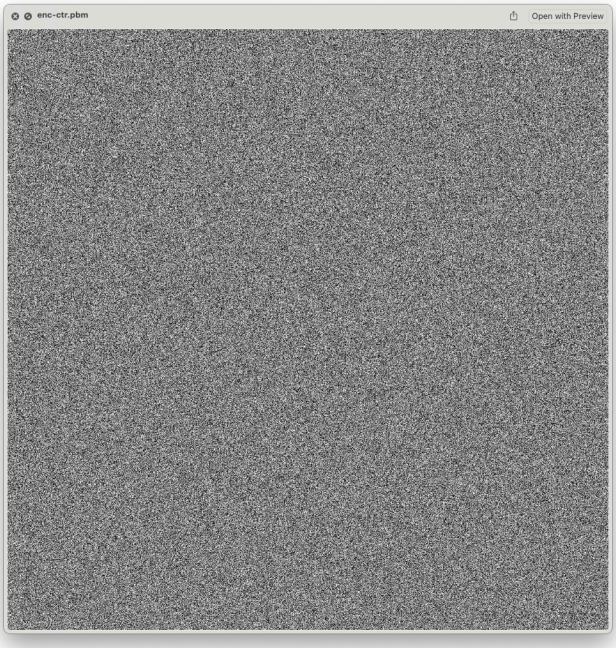
d. Pad the header back and see the result.
$ cp enc.x enc.pbm
$ vi enc.pbm
P4
2000 2000
KR)B@HD@

Answer:



e. You may try it with other modes with IV, chaining, or feedback and compare the result.

Answer:
ECB: Large-scale patterns/silhouettes remain visible (identical blocks → identical ciphertext blocks).
CBC / CTR / CFB / OFB: Image appears noise-like; no recognizable structure.
- CBC uses chaining (each block depends on the previous ciphertext block + IV).
- CTR/CFB/OFB turn the block cipher into a keystream; identical plaintext blocks at different positions encrypt differently (thanks to IV/counter state).



Img. AES-256-CTR

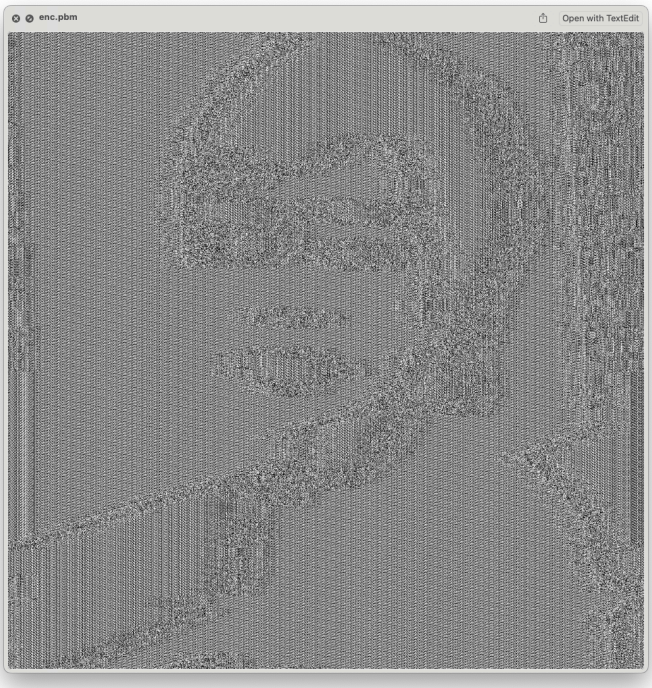
f. What does the result suggest about the mode of operation in block cipher? Please provide your analysis.

Answer:

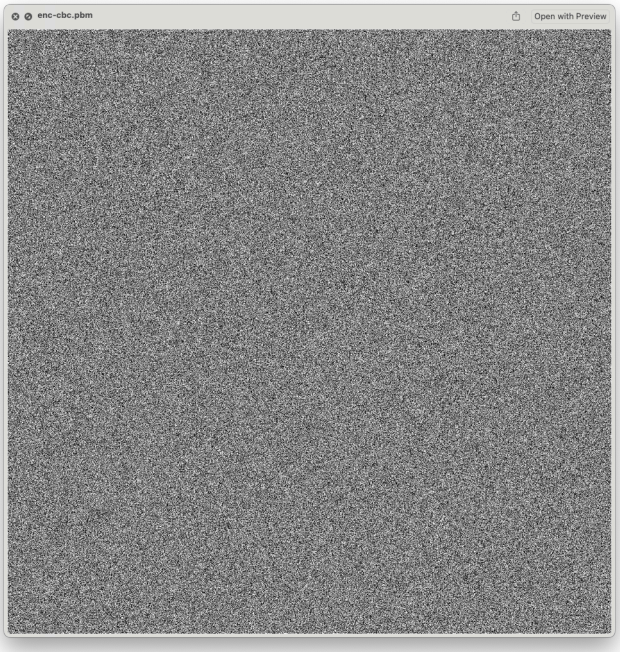
The experiment shows that mode of operation determines whether a block cipher actually hides structure: ECB fails (patterns survive), while modes with IV/chaining/keystream (CBC/ CTR/CFB/OFB) remove visible patterns—so in practice, use AEAD modes and handle IVs/nonces correctly.

Img. Original

Img. AES-256-ECB



Img. AES-256-CBC



1) ECB is insecure for structured data

- In the ECB image, large shapes and textures are still visible.
- Reason: **identical plaintext blocks → identical ciphertext blocks** (deterministic, no randomness, no inter-block dependency).

- Consequence: ECB **leaks patterns** and **position-dependent structure**. It provides confidentiality only for "random-looking" data and is **not semantically secure**. Do **not** use ECB for real data.

2) Modes with IV/chaining/feedback hide patterns

- In CBC/CTR/CFB/OFB, the encrypted images look like noise.

- Why: an **IV (initialization vector)** and **inter-block dependence** (or a keystream) ensure that **equal plaintext blocks encrypt differently**.

- Result: patterns disappear; the ciphertext reveals far less about the plaintext's structure.

3) Correct use of IVs matters

- IV must be **unpredictable/unique** (mode-dependent):

  - CBC needs **random IV** per message.

  - CTR/OFB/CFB need **unique nonces** (never reuse with same key).

- Reusing an IV/nonce with the same key **re-exposes patterns** or enables serious attacks (e.g., keystream reuse in CTR/OFB/CFB).

4) Error propagation & parallelism differ by mode

- **ECB/CTR**: per-block independence → easy to parallelize; single-bit errors affect only that block/bit.

- **CBC**: decryption of a block depends on the previous ciphertext block → less parallelism; a bit error corrupts two blocks.

- **CFB/OFB/CTR**: stream-like; **no padding** needed and errors don't cascade, but nonce/IV misuse is critical.

5) Confidentiality ≠ integrity

- These modes (ECB/CBC/CTR/CFB/OFB) provide **encryption only**. They do **not** detect tampering.

- For real systems, prefer **AEAD** modes (e.g., **GCM**, **CCM**) that combine encryption with authentication to prevent bit-flipping or cut-and-paste attacks.

6) Practical takeaway / recommendations

- **Never use ECB** for anything beyond lab demos.

- Use **AES-GCM** (or another AEAD mode) by default; if you must use **CBC**, use a **random IV** and add a **MAC** (e.g., HMAC) over the IV+ciphertext.

- For CTR/CFB/OFB, **never reuse nonces**, and treat them like "one-time counters" per key.

Q4.

(Encryption Protocol - Digital Signature)

a. Measure the performance of a hash function (sha1), RC4, Blowfish and DSA. Outline your experimental design. (Please use OpenSSL for your measurement)

Answer:

Objective

The goal is to measure and compare the performance of different cryptographic primitives — a hash function (**SHA-1**), a stream cipher (**RC4**), a block cipher (**Blowfish in ECB and CBC modes**), and a digital signature scheme (**DSA-2048**) — using **OpenSSL**. This allows us to observe differences in throughput and computational cost between symmetric, hashing, and asymmetric operations.

Environment Setup

- **System Information**: CPU model, OS version, and OpenSSL build/version were recorded to ensure reproducibility.
- **Consistency**: Experiments were conducted under similar conditions (AC power, minimal background load).
- **Providers**: On OpenSSL 3.x, the **legacy provider** was explicitly enabled to allow benchmarking of RC4 and Blowfish.

Measurement Methodology

1. **Tool**: The built-in OpenSSL benchmarking utility openssl speed was used with the -elapsed option.
2. **Duration**: Each algorithm was run for ~3–10 seconds per trial.
3. **Trials**: Each algorithm was tested in **three independent trials**; the **median** value was taken to reduce noise.
4. **Data Points**:
    - For **SHA-1, RC4, and Blowfish**, throughput was measured in **MB/s**, using the largest block size (**16384 bytes**) as the stable indicator of maximum throughput.
    - For **DSA-2048**, performance was measured in **operations per second (ops/s)**, for both **signing** and **verification**.

Parameters and Commands

- **SHA-1**: openssl speed -evp sha1
- **RC4**: openssl speed -evp rc4 -provider legacy

- **Blowfish**: openssl speed -evp bf-ecb and openssl speed -evp bf-cbc
- **DSA-2048**: openssl speed dsa2048

Output Handling

- Raw logs (*.log) were collected for each run.
- A parsing script summarized results into summary.csv, with medians across trials.
- For analysis, the **16384-byte throughput line** was used for symmetric and hashing algorithms, while DSA results came directly from the reported sign/s and verify/s lines.

Reproducibility

- The entire procedure was automated in a single benchmarking script (openssl_bench.sh).
- Results are archived in timestamped folders, including raw logs, summary CSV, and environment details.
- Any user with the same hardware and OpenSSL version can reproduce the experiment.

b. Comparing performance and security provided by each method.

Answer:

Performance

- **SHA-1 (hashing)**: Achieved the highest throughput (~2.6 GB/s). This is expected because hashing is optimized for bulk data processing and does not require key management or block chaining.

- **RC4 (stream cipher)**: Performed very fast (~1.17 GB/s), close to SHA-1. Its simple XOR-based design makes it computationally light.

- **Blowfish (block cipher)**: Slower than RC4, with ~355 MB/s in ECB mode and ~152 MB/s in CBC mode. Block ciphers require more complex key schedules and per-block transformations, and CBC adds dependency between blocks, further reducing throughput.

- **DSA-2048 (digital signature)**: By far the slowest — only ~5,500 signatures/s and ~6,100 verifications/s. Asymmetric cryptography is orders of magnitude more expensive because it relies on modular exponentiation over large integers, unlike the lightweight operations in hashing or symmetric ciphers.

Relative ranking (fastest → slowest):
SHA-1 > RC4 > Blowfish > DSA

Security

- **SHA-1**: Considered **broken** due to practical collision attacks (e.g., SHAttered in 2017). Still fast, but unsuitable for modern cryptographic integrity. Modern alternatives: SHA-256, SHA-3.

- **RC4**: **Deprecated and insecure**. It has statistical biases that enable practical attacks (notably against WEP and old TLS). No longer recommended.

- **Blowfish**: Still unbroken, but has a **64-bit block size**, which is unsafe for large amounts of data (susceptible to birthday attacks, e.g., SWEET32). Also, AES is faster and more widely supported today.

- **DSA-2048**: Provides **secure digital signatures** if implemented correctly. However, it depends critically on the randomness of the per-message nonce k; weak or reused k values can expose the private key. While secure at 2048-bit, it is slower and less commonly deployed than RSA-PSS or elliptic-curve signatures (ECDSA, Ed25519).

Trade-off Summary

- **Hash and symmetric algorithms** (SHA-1, RC4, Blowfish) are optimized for **speed**, but vary widely in **security strength**: SHA-1 and RC4 are obsolete, Blowfish is marginal.

- **Asymmetric signatures (DSA)** provide **authentication and non-repudiation**, but at much higher computational cost. They are secure when parameters and randomness are strong, but much less efficient than symmetric primitives.

Bottom line: SHA-1 and RC4 are fast but insecure; Blowfish is moderately secure but dated; DSA is secure for signatures but slow. Modern systems favor **SHA-256/3 + AES (symmetric)** for performance and **ECDSA/Ed25519 or RSA-PSS** for digital signatures.

c. Explain the mechanism underlying Digital Signature. How does it combine the strength and weakness of each encryption scheme?
Hint: (OpenSSL command line)
# List algorithms
$ openssl list cipher-algorithms
# To encrypt
$ openssl enc -ciphername [options] -e -in filename -out filename \ -K key -iv IV -nopad -nosalt

Answer:

A **digital signature** is a cryptographic method that provides **integrity, authenticity, and non-repudiation** for digital data. Its mechanism combines a **hash function** with an **asymmetric key algorithm**:

1. **Hashing the message**:
    - Instead of signing the entire message (which could be very large), the sender first applies a **hash function** (e.g., SHA-256) to produce a fixed-length digest.
    - This makes the process efficient and ensures that any small change in the message results in a completely different digest.
2. **Signing the digest**:
    - The sender uses their **private key** (DSA, RSA, or ECDSA) to sign the hash digest.
    - The output is the **digital signature**, which is unique to both the message content and the signer's private key.
3. **Verification**:
    - The receiver recomputes the hash of the received message.
    - Using the sender's **public key**, the receiver verifies that the signature corresponds to the recomputed digest.
    - If they match, the message is authentic and unaltered.

How It Combines Strengths and Weaknesses

- **Hash function strength**:
    - Very **fast** to compute, handles arbitrary-length input, and detects small changes.

- But on its own, a hash cannot prove *who* created the message — only that the message is unchanged.

- **Asymmetric encryption strength**:
  - Provides **authentication** (only the holder of the private key could generate the signature) and **non-repudiation** (the signer cannot later deny creating it).
  - But asymmetric operations are **slow** and inefficient for large data.

- **Combination (Digital Signature)**:
  - By hashing first, the system avoids the inefficiency of signing large data with an asymmetric algorithm.
  - By signing the digest, it prevents forgery and ties the signature to the signer's private key.
  - The weakness of each primitive (hash alone lacks authentication, asymmetric alone is too slow) is compensated by the other.
  - **Caution**: if the hash is weak (e.g., SHA-1 collisions) or if the signature algorithm is misused (e.g., poor randomness in DSA), the whole scheme is compromised. Thus, both components must be strong.

In summary:
A digital signature is a hybrid approach: it leverages the **speed and sensitivity of hashing** with the **security and authenticity of public-key cryptography**, creating a mechanism that is both practical and secure.