Computer Security

Assignment 4, Basic Encryption

Yanapat Patcharawiwatpong

1 Introduction

Cryptography serves as the cornerstone of modern computer security, ensuring data confidentiality, integrity, and authenticity. This assignment introduces core concepts through hands-on exercises with Caesar and Vigenère ciphers, AES encryption modes, and digital signatures. Additionally, this task evolves analyze encryption vulnerabilities and the importance of secure block cipher modes. Using tools like OpenSSL and ImageMagick, this activity provides practical experience in implementing and evaluating cryptographic algorithms, offering insights into both historical and modern encryption methods.

2 Exercise

2.1 Encryption and Statistical Analysis

2.1.1 Frequency of top three most frequent letters

The Python script analyzes letter frequencies in the cipher text using ‘collections.Counter‘ to count occurrences and ‘PrettyTable‘ to format the results. It filters out non-alphabetic characters, determines the top three frequencies, and organizes the data into a table showing the rank, characters, and their frequencies. This method effectively highlights the most frequent letters for cryptographic analysis.

```python
from collections import Counter
from prettytable import PrettyTable

cipher_text = "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."
filtered_text = ''.join(filter(str.isalpha, cipher_text))
freq_count = Counter(filtered_text)

top_counts = sorted(set(freq_count.values()), reverse=True)[:3]
freq_groups = {count: ', '.join(sorted(char for char, c in freq_count.items() \
    if c == count)) for count in top_counts}

table = PrettyTable(["Rank", "Characters", "Frequency"])
table.add_rows([(i+1, freq_groups[count], count) for i, count in enumerate(top_counts)])

table
```

The frequency analysis reveals that ’P’ is the most frequent character, appearing 7 times, followed by ’F’, ’O’, and ’R’ with 6 occurrences each, and ’Q’ with 5. This distribution highlights ’P’ as a key character and suggests that 'F', 'O', and 'R' are also significant. The lower frequency of 'Q' may indicate its less frequent use in the cipher. These findings can help identify patterns and guide further decryption efforts.

2.1.2 Commonly Used Words and Decryption Hints

In English, commonly used words can provide crucial hints in decryption. The following table lists frequently occurring two-letter and three-letter words, which are often used in substitution ciphers.

Word Length	**Common Words**
2-Letter Words	as, at, be, he, if, in, is ,it, of, to
3-Letter Words	and, are, but, for, get, not, one, the, too, was

Table 1: Commonly Used Two-Letter and Three-Letter Words in English

Recognizing these words within the ciphertext can help identify letter substitutions and facilitate the decryption process by providing clues about common patterns in the text.

2.1.3 Time Taken for Decryption

The process of cracking the given cipher text was completed within a time frame of approximately 25 to 30 minutes. This duration includes the time spent analyzing character frequencies, identifying common patterns, and applying decryption techniques. The relatively short time required reflects the effectiveness of frequency analysis and pattern recognition in deciphering the message.

2.1.4 Decryption Methodology

The decryption of this cipher can be approached using techniques that combine traditional cryptanalysis methods with computational efficiency and linguistic analysis. The process involves a structured examination of letter frequency, common word patterns:

1. **Identifying Common Short Words Using Frequency Analysis**: The first step is to analyze the ciphertext for repeating patterns and common word structures. In the given cipher, the sequence "FP" appears multiple times. Based on frequency analysis and typical two-letter word usage in English, it is reasonable to hypothesize that "FP" could correspond to the word "is." Using this assumption, thus, deduce that "F = i" and "P = s."

2. **Word Length and Contextual Clues**: A key technique used is to leverage known structures of the English language, such as the length of words and the positions of common letters. The word "PRCSOFQX" consists of eight letters and starts with "S" (since "P = s" was previously deduced). Given the structure and context, a plausible guess is that the word could be "Security." This leads to the following character mappings:

    - "R = e"
    - "C = c"
    - "S = u"
    - "O = r"
    - "Q = t"
    - "X = y"

3. **Other Word Frequencies and Pattern Recognition**: Next key step is to analyze character frequencies and compare them against typical English letter distributions. By observing common three-letter sequences like "QDR," it is likely that this sequence represents the word "the," which leads to the conclusion that "D = h."

   Similarly, the sequence "AFOPQ" fits the pattern "_irst," indicating it could be the word "first," resulting in "A = f." The word "CZSPR" follows the pattern for "cause," suggesting "Z = a."

   With this method, "LA" aligns with "of," providing "L = o." Further analysis reveals that "JFPALOQSKR" corresponds to "misfortune," giving the additional mappings:

   - "J = m"
   - "K = n"

4. **Contextual Guessing for Longer Words**: Based on context and known language patterns, the next part of the message reads: "Security is the first cause of misfortune." then, analyze the remaining words. The sequence "ZK" fits the structure of "an," while "LIU" corresponds to "old." The phrase "BROJZK" matches the word "German," resulting in the additional mapping of "B = g."

   Finally, "MOLTROE" fits the pattern for "proverb," completing the remaining letter mappings:

   - "M = p"
   - "T = v"
   - "E = b"

5. **Final Decoded Message**: By applying these mappings and leveraging both manual and automated techniques, the final decrypted message is:

   Security is the first cause of misfortune. This is an old German proverb.

2.1.5 Decryption Speed with Knowledge of Caesar Cipher

Recognizing that the encryption is a Caesar cipher, a monoalphabetic substitution cipher, significantly accelerates decryption. The Caesar cipher shifts each letter by a fixed number of positions, limiting possible shifts to 25. This makes it highly susceptible to frequency analysis or brute-force testing. Knowing the cipher type allows the cryptanalyst to quickly test all possible shifts or apply frequency analysis, greatly reducing the time needed to crack the message.

2.1.6 Cipher Disc Representation

To understand the monoalphabetic substitution cipher used in the given text, which can represent the cipher as a disc. The disc shows the correspondence between the plaintext and ciphertext alphabets as follows:

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**
Z	*E*	*C*	*U*	*R*	*A*	*B*	*D*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*S*	*T*	*V*	*W*	*X*	*Y*

The cipher disc visually represents the relationship between plaintext and ciphertext alphabets, facilitating the decryption process. Each letter in the plaintext alphabet is mapped to a corresponding letter in the ciphertext alphabet. This systematic representation aids in decrypting messages by providing a clear view of the substitution pattern used, making the decryption process more efficient.

2.1.7 Brute-Force Decryption of Caesar Cipher

To decrypt a Caesar cipher, a brute-force approach can be employed, wherein every possible shift is tested to ascertain the correct plaintext. The Python program detailed here applies this methodology, incorporating natural language processing techniques to enhance decryption accuracy. By leveraging English word lists and frequency analysis, the program systematically evaluates potential plaintexts to identify the most probable decryption. The integration of linguistic resources allows for the assessment of decrypted messages based on their alignment with expected English word patterns and letter frequencies.

```python
import nltk
from nltk.corpus import words, brown
from nltk.probability import FreqDist
from nltk import ngrams
import re
from functools import lru_cache
from tqdm import tqdm
from IPython.display import clear_output

nltk.download('words', quiet=True)
nltk.download('brown', quiet=True)

nltk_words = set(word.lower() for word in words.words())
brown_words = set(word.lower() for word in brown.words())
word_freq = FreqDist(word.lower() for word in brown.words())

bigram_freq = FreqDist(ngrams(brown.words(), 2))
frequency_order = 'etaoinshrdlcumwfgypbvkjxzq'[::-1]

@lru_cache(maxsize=1000)
def create_pattern(substitution_pattern):
    return ''.join('*' if char.isupper() else char for char in substitution_pattern)

@lru_cache(maxsize=1000)
def find_matching_words(substitution_pattern):
    pattern = create_pattern(substitution_pattern)
    regex_pattern = re.compile(f'^{pattern.replace("*", ".")}$')
    return sorted(
        (word for word in nltk_words if regex_pattern.fullmatch(word)),
        key=lambda x: word_freq[x],
        reverse=True
    )

@lru_cache(maxsize=1000)
def validate_mapping(cipher, decipher):
    return len(set(cipher)) == len(set(decipher))

def evaluate_decryption(decrypted_message):
    words = decrypted_message.split()
    message_bigrams = list(ngrams(words, 2))
    bigram_score = sum(bigram_freq.get(bigram, 0) for bigram in message_bigrams)
    english_word_count = sum(1 for word in words if word.lower().replace(".", "") \
        in brown_words)
    english_word_ratio = english_word_count / len(words) if words else 0
    char_score = sum(frequency_order.index(char.lower()) for char in decrypted_message \
        if char.isalpha())
    return bigram_score, english_word_ratio, char_score

def solve(text):
    chunks = sorted(set(text.replace(".", "").split(" ")), key=len, reverse=True)

    def iterate_subs(chunk, index, mapping):
        if all(word.islower() for word in chunk):
            if not validate_mapping("".join(chunks), "".join(chunk)):
                return []
            copy_text = text
            for i, word in enumerate(chunk):
                copy_text = copy_text.replace(chunks[i], word)
            return [copy_text]

        if index >= len(chunk):
            return []

        if chunk[index].islower():
            return iterate_subs(chunk, index + 1, mapping)

        matches = find_matching_words(chunk[index])
        if not matches:
            return []

        possible_answers = []
        for match in tqdm(matches, desc=f"Processing chunk: ", leave=False):
            new_mapping = mapping.copy()
            valid = True
            for c, m in zip(chunk[index], match):
                if c.isupper():
                    if c in new_mapping and new_mapping[c] != m:
                        valid = False
                        break
                    new_mapping[c] = m
            if not valid:
                continue

            subbed_chunk = [word.translate(str.maketrans(new_mapping)) for word in chunk]
            possible_answers.extend(iterate_subs(subbed_chunk, index + 1, new_mapping))

        clear_output(wait=True)

        return possible_answers

    print("Solving cipher...")
    possible_texts = iterate_subs(chunks, 0, {})
    best_decryption = None
    best_score = float('-inf')

    for decrypted_message in tqdm(possible_texts, desc="Evaluating decryptions", \
        unit="text", leave=False):
        bigram_score, english_word_ratio, char_score = evaluate_decryption(decrypted_message)
        if english_word_ratio == 1:
            score = bigram_score + char_score
            if score > best_score:
                best_score = score
                best_decryption = decrypted_message

    return best_decryption

cipher = "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."
result = solve(cipher)
print("Decrypted message:", result.upper())
```

1. **Preparation of Resources:** The code utilizes the Natural Language Toolkit (nltk) to obtain English word lists and bigram frequency data from the Brown corpus. These resources are essential for validating decrypted texts. Words are preprocessed and stored in sets for efficient lookup, while frequencies of words and bigrams are computed for evaluation purposes.

2. **Pattern Matching:** The function `find_matching_words()` employs regular expressions to match candidate plaintext words against a predefined list of English words. This matching process is based on patterns derived from the ciphertext, which assists in identifying plausible decryptions.

3. **Validation and Evaluation:** The `validate_mapping()` function ensures that substitutions preserve a one-to-one relationship between ciphertext and plaintext characters. The `evaluate_decryption()` function assesses decrypted messages based on bigram frequencies, the proportion of valid English words, and alignment with expected letter distributions.

4. **Brute-Force Attack:** The `solve()` function implements the brute-force decryption approach by exploring all possible mappings and assessing their validity. This recursive method iterates through potential word substitutions, refining mappings based on previously identified patterns.

5. **Output and Results:** The code evaluates all possible decrypted texts, ranking them according to their linguistic coherence and alignment with English letter frequencies. The final output is the decrypted message with the highest score based on combined metrics of bigram frequency and character distribution.

The code demonstrates an effective brute-force approach to decrypting Caesar ciphers by integrating pattern matching, frequency analysis, and bigram scoring. The use of the nltk library significantly aids in validation and scoring through extensive linguistic data. Despite its robustness, the method can be computationally intensive, particularly for larger or more complex ciphertexts. The success of this decryption approach depends on the quality of the linguistic resources and frequency data used. While brute-force techniques ensure eventual decryption, they may not always be the most efficient for more sophisticated ciphers or larger datasets. Overall, the code provides a practical application of cryptographic techniques and natural language processing for deciphering simple substitution ciphers.

2.2 Vigenère Cipher Analysis

The Vigenère cipher is an advanced form of the Caesar cipher that employs polyalphabetic substitution. Unlike the Caesar cipher, which uses a single substitution rule for all letters, the Vigenère cipher utilizes a sequence of Caesar ciphers determined by the letters of a keyword. This approach introduces variability in the encryption process, enhancing security against frequency analysis attacks.

2.2.1 Encryption Mechanism

The Vigenère cipher encrypts data by applying a series of Caesar ciphers determined by the letters of a keyword. For each letter in the plaintext, the corresponding letter in the keyword is used to shift the plaintext letter. Specifically, the shift amount for each letter is determined by its position in the alphabet relative to the keyword letter. This results in a sequence of shifts that vary throughout the message, making the encryption more resistant to simple frequency analysis.

2.2.2 Security Analysis

The Vigenère cipher, when compared to the Caesar cipher, provides significantly improved security. While the Caesar cipher is vulnerable to frequency analysis due to its fixed shift value, the Vigenère cipher's use of a variable shift based on the keyword mitigates this vulnerability.

For instance, if the key is the word "CAT":

- The letter 'C' corresponds to a shift of 2 (since 'C' is the third letter of the alphabet, starting from 0).
- The letter 'A' corresponds to a shift of 0.
- The letter 'T' corresponds to a shift of 19.

The use of a repeating keyword ensures that each letter in the plaintext is shifted by a different amount, depending on the keyword. This creates a more complex pattern of shifts that is less susceptible to frequency analysis compared to the Caesar cipher, which uses a single uniform shift value.

However, the security of the Vigenère cipher is not invulnerable. For a sufficiently long and repeated keyword, patterns may still be discernible, especially if the keyword is short relative to the plaintext. Cryptanalysts can use techniques such as the Kasiski examination and Friedman test to attack the cipher.

2.2.3 Python Implementation

The following Python program demonstrates the Vigenère cipher for encryption and decryption:

```python
def vigenere(text: str, key: str, encrypt=True) -> str:
    key = key.upper()
    len_key = len(key)
    shift = lambda c, k: chr((ord(c) - ord('A') + k) % 26 + ord('A'))
    return ''.join(
        shift(c, ord(key[i % len_key]) - ord('A'))
        if c.isalpha() else c
        for i, c in enumerate(text.upper())
    ) if encrypt else ''.join(
        shift(c, - (ord(key[i % len_key]) - ord('A')))
        if c.isalpha() else c
        for i, c in enumerate(text.upper())
    )

def vigenere_encrypt(text: str, key: str) -> str:
    return vigenere(text, key, True)

def vigenere_decrypt(text: str, key: str) -> str:
    return vigenere(text, key, False)

# Example usage
KEY = "CAESAR"
TEXT = "ALEA IACTA EST"

encrypted = vigenere_encrypt(TEXT, KEY)
decrypted = vigenere_decrypt(encrypted, KEY)

print(f"Encrypted: {encrypted}") # CLIS ZCCXS VUT
print(f"Decrypted: {decrypted}") # ALEA IACTA EST
```

The provided Python code implements the Vigenère cipher for both encryption and decryption. The `vigenere` function, central to this implementation, processes text using a keyword to determine shift values, adjusting each character accordingly. It accepts three parameters: the text, the key, and a boolean flag indicating the operation mode (encryption or decryption). The `vigenere_encrypt` and `vigenere_decrypt` functions act as wrappers to facilitate these operations. The example demonstrates encrypting *ALEA IACTA EST* with the key *CAESAR* and decrypting it back to the original plaintext.

This implementation highlights the Vigenère cipher's improved security over the Caesar cipher through its use of keyword-based shifting. The code is clear and utilizes Python's string and lambda functions efficiently. While suitable for educational purposes and basic encryption tasks, it is recommended to use more advanced cryptographic algorithms for enhanced security in practical applications.

2.3 Block Cipher Mode Evaluation

The task was addressed using a combination of tools and Python code. ImageMagick was utilized to convert the images to PBM format, and OpenSSL was used for encryption, with commands executed through Python. The Python code automates the entire process: converting the image to PBM format, removing the header, performing encryption with different modes, and converting the encrypted PBM back to PNG. This approach effectively demonstrates the impact of block cipher modes on image encryption.

```python
import io
import subprocess
from PIL import Image
import matplotlib.pyplot as plt

MAGICK = "/home/linuxbrew/.linuxbrew/bin/magick" # Path to magick bin
FILE = "AQUARA.png" # Picture used
```

```

def run_command(command, input_data=None):
    result = subprocess.run(command, shell=True, capture_output=True, input=input_data)
    return io.BytesIO(result.stdout)

def convert_to_pbm(image_path, size="2000x2000"):
    return run_command(f"{MAGICK} {image_path} -resize {size} pbm:-")

def remove_pbm_header(pbm_file):
    pbm_data = pbm_file.read()
    return io.BytesIO(b'\n'.join(pbm_data.split(b'\n')[2:]))

def encrypt_image(input_file, algo, passwd=" "):
    command = f"openssl enc {algo} -in /dev/stdin -nosalt -pass pass:{passwd} -out /dev/stdout"
    return run_command(command, input_file.read())

def add_pbm_header(encrypted_file):
    return io.BytesIO(b"P4\n2000 2000\n" + encrypted_file.read())

def encode(image, algo):
    pbm_file = convert_to_pbm(image)
    raw_file = remove_pbm_header(pbm_file)
    encrypted_file = encrypt_image(raw_file, algo)
    return Image.open(io.BytesIO(add_pbm_header(encrypted_file).read()))

def main():
    original_img = Image.open(FILE)
    ecb_img = encode(FILE, "-aes-256-ecb")
    cbc_img = encode(FILE, "-aes-256-cbc")

    fig, axes = plt.subplots(1, 3, figsize=(15, 5))
    axes[0].imshow(original_img)
    axes[0].set_title("Original")
    axes[0].axis('off')

    axes[1].imshow(ecb_img)
    axes[1].set_title("AES-256-ECB")
    axes[1].axis('off')

    axes[2].imshow(cbc_img)
    axes[2].set_title("AES-256-CBC")
    axes[2].axis('off')

    plt.tight_layout()
    plt.show()

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

The Python code effectively demonstrates the impact of different block cipher modes on image encryption. It first converts the image to PBM format using ImageMagick, removes the header to isolate the pixel data, then performs encryption using OpenSSL with the specified modes. The header is then restored, and the result is converted back to PNG format. This automated process provides a clear visual comparison of the original image and its encrypted forms.

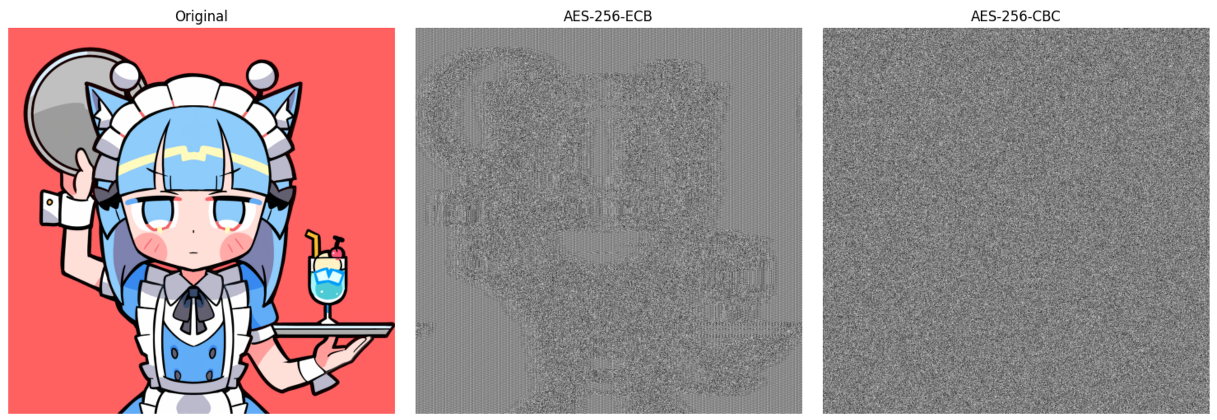

Figure 1: Comparison of image encryption: Original, AES-256-ECB, and AES-256-CBC.

The comparison between AES-256-ECB and AES-256-CBC encryption modes reveals significant differences. AES-256-ECB mode retains recognizable features of the original image, as identical plaintext blocks are encrypted into identical ciphertext blocks, preserving spatial patterns. In contrast, AES-256-CBC mode, which uses an initialization vector (IV) and a chaining mechanism, introduces randomness that obscures the original image's content, making it unrecognizable. This demonstrates that ECB mode's lack of feedback allows for pattern retention, while CBC mode enhances data confidentiality by incorporating feedback and IV.

2.4 Performance of Encryption Protocol

2.4.1 Algorithm Performance Measurements

The performance evaluation of various cryptographic algorithms—SHA-1, RC4, Blowfish, and the Digital Signature Algorithm (DSA)—was conducted using OpenSSL. The experimental process involved:

1. **File Generation:** Random files of varying block sizes (16 bytes to 16,384 bytes) were created.
2. **Measurement:** Execution times were recorded for SHA-1 hashing, RC4 and Blowfish encryption, and DSA signing.
3. **Data Collection:** Average execution times were computed from 10 trials for each block size.

The measurement process utilized the following Python code:

```python
import os
import time
import subprocess
import tempfile
import csv
from tqdm import tqdm
```

```python

OPENSSL = "/home/linuxbrew/.linuxbrew/opt/openssl@1.1/bin/openssl" # Path to Openssl@1.1 bin
BLOCK_SIZES = [16, 64, 256, 1024, 4096, 16384]
TRIALS = 10

def generate_random_file(block_size, file_size_gb=1):
    file_size = int(file_size_gb * 1024 * 1024 * 1024)
    with tempfile.NamedTemporaryFile(delete=False) as temp_file:
        file_name = temp_file.name
        print(f"Generating {file_size_gb} GB file: {file_name}")

        while file_size > 0:
            temp_file.write(os.urandom(min(block_size, file_size)))
            file_size -= block_size

    return file_name

def measure_openssl(command):
    start = time.time()
    result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, text=True)
    elapsed = time.time() - start
    if result.returncode == 0:
        return elapsed
    else:
        raise RuntimeError("Command failed")

def measure_sha1(file_name):
    elapsed = measure_openssl(f"{OPENSSL} sha1 {file_name}")
    return elapsed

def measure_rc4(file_name):
    key = os.urandom(16).hex()
    elapsed = measure_openssl(f"{OPENSSL} enc -rc4 -in {file_name} -out /dev/null -k {key} -pbkdf2")
    return elapsed

def measure_blowfish(file_name):
    key = os.urandom(16).hex()
    elapsed = measure_openssl(f"{OPENSSL} enc -bf -in {file_name} -out /dev/null -k {key} -pbkdf2")
    return elapsed

def measure_dsa(file_name):
    subprocess.run(f"{OPENSSL} dsaparam -out /tmp/dsa_params.pem 1024", shell=True, check=True, stderr=subprocess
    subprocess.run(f"{OPENSSL} gendsa -out /tmp/dsa_key.pem /tmp/dsa_params.pem", shell=True, check=True, stderr=

    sign_command = f"{OPENSSL} dgst -sha1 -sign /tmp/dsa_key.pem -out /dev/null {file_name}"
    sign_elapsed = measure_openssl(sign_command)

    os.remove("/tmp/dsa_params.pem")
    os.remove("/tmp/dsa_key.pem")

    return sign_elapsed
def perform_trial(block_size):
    print(f"Performing trial with block size {block_size} bytes...")

    sha1_times = []
    rc4_times = []
    blowfish_times = []
    dsa_sign_times = []

    for _ in tqdm(range(TRIALS), desc=f"Trials for block size {block_size}"):
        file_name = generate_random_file(block_size=block_size)

        sha1_times.append(measure_sha1(file_name))
        rc4_times.append(measure_rc4(file_name))
        blowfish_times.append(measure_blowfish(file_name))
        dsa_sign_times.append(measure_dsa(file_name))

        os.remove(file_name)

    return {
        'sha1_avg': sum(sha1_times) / TRIALS,
        'rc4_avg': sum(rc4_times) / TRIALS,
        'blowfish_avg': sum(blowfish_times) / TRIALS,
        'dsa_sign_avg': sum(dsa_sign_times) / TRIALS,
    }

def write_results_to_csv(results, filename='algorithm_performance.csv'):
    with open(filename, 'w', newline='') as csvfile:
        fieldnames = ['Block Size', 'SHA1 Avg Time', 'RC4 Avg Time', 'Blowfish Avg Time', 'DSA Sign Avg Time']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for block_size, times in results.items():
            writer.writerow({
                'Block Size': block_size,
                'SHA1 Avg Time': times['sha1_avg'],
                'RC4 Avg Time': times['rc4_avg'],
                'Blowfish Avg Time': times['blowfish_avg'],
                'DSA Sign Avg Time': times['dsa_sign_avg']
            })

def main():
    results = {}
    for block_size in tqdm(BLOCK_SIZES, desc="Block Sizes"):
        avg_times = perform_trial(block_size)
        results[block_size] = avg_times

    write_results_to_csv(results)

```

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

The provided Python code evaluates the performance of various cryptographic algorithms using OpenSSL. It generates random files of different block sizes and measures the execution times for SHA-1 hashing, RC4 and Blowfish encryption, and DSA signing. Performance metrics are collected over multiple trials, averaged, and then written to a CSV file. The main part of the code orchestrates the execution of trials for different block sizes and manages the collection of results.

The results of the performance evaluation are summarized in Table 2.

Block Size (bytes)	**SHA1 Avg Time (s)**	**RC4 Avg Time (s)**	**Blowfish Avg Time (s)**	**DSA Sign Avg Time (s)**
16	0.452	1.619	4.544	0.453
64	0.450	1.607	4.520	0.451
256	0.456	1.619	4.536	0.456
1024	0.456	1.618	4.545	0.457
4096	0.453	1.610	4.531	0.451
16384	0.453	1.613	4.517	0.455

Table 2: Average execution times for various cryptographic algorithms with different block sizes.

2.4.2 Comparing Performance and Security

The performance analysis of the cryptographic algorithms reveals distinct differences:

- **SHA-1:** This algorithm consistently exhibited the fastest execution times across all block sizes. However, while it is efficient, SHA-1 is known to have vulnerabilities and is generally considered less secure for modern applications.
- **RC4:** RC4 demonstrated faster performance than Blowfish but was slower than SHA-1. Despite its speed advantage over Blowfish, RC4 is outdated and has been found to have several security weaknesses.
- **Blowfish:** This algorithm was the slowest in the tests but provided strong encryption. Blowfish is more secure than RC4 but comes at the cost of performance, making it less suitable for applications requiring high-speed encryption.
- **DSA:** The signing times for DSA were comparable to those of SHA-1 but showed some variation with block size. DSA combines the efficiency of SHA-1 for digest generation with robust security for signing. However, its signing operation is generally slower compared to direct hashing methods.

These observations underscore the trade-offs between performance and security in cryptographic protocols. SHA-1, while the fastest, has known security issues. RC4, although faster than Blowfish, is no longer recommended due to security vulnerabilities. Blowfish, while secure, is slower, and DSA offers a balance between security and performance but with variable speeds depending on the block size.

2.4.3 Mechanism of Digital Signatures

Digital signatures utilize a combination of hashing and encryption to ensure the integrity and authenticity of data. The process involves:

1. **Hashing:** A hash function, such as SHA-1, generates a fixed-size digest of the data. This digest acts as a unique fingerprint of the data.
2. **Signing:** The generated hash digest is encrypted using a private key (e.g., with DSA), producing a digital signature. This signature is unique to both the digest and the private key used for signing.
3. **Verification:** The recipient uses the sender's public key to decrypt the signature and retrieve the hash digest. They then compute the hash of the received data and compare it with the decrypted digest. A match confirms the integrity of the data and the authenticity of the sender.

Digital signatures effectively leverage the strengths of both hashing and encryption. Hash functions provide a fast and efficient means of creating a digest, while encryption ensures secure signing and verification. This approach allows for the detection of any data tampering and verifies the sender's identity, addressing the inherent trade-offs between the speed of hashing and the security of encryption.