Lab 2

1. Introduction

This lab aimed to demonstrate the inherent weaknesses in classic cryptographic systems by practically attacking and decrypting ciphertexts. We targeted two primary systems: the **Caesar Cipher** (Checkpoint 1) and a **Mono-alphabetic Substitution Cipher** (Checkpoint 2). The objective was not just to decrypt the messages but to understand the methodologies and thought processes involved in breaking these ciphers.

2. Checkpoint 1: Breaking the Caesar Cipher

2.1. Ciphertext

`odrobcewscdrolocdewkbdmyxdbkmdzvkdpybwyeddrobo`

2.2. Approach and Methodology

The Caesar cipher is a mono-alphabetic substitution cipher where each letter in the plaintext is shifted a fixed number of places down or up the alphabet. The limited key space (only 25 possible shifts for English) makes it highly vulnerable to a **Brute-Force Attack**.

The approach taken was systematic:

1. **Brute-Force Implementation:** A program was written in Python that took the ciphertext as input.
2. **Algorithm:** The program iterated through all possible shifts from 1 to 25. For each shift `k`, it decrypted the ciphertext by shifting each alphabetic character backwards by `k` positions in the alphabet, wrapping around from `a` to `z` as necessary.
3. **Manual Analysis:** The output for all 25 shifts was printed. The goal was to visually inspect each resulting plaintext candidate and identify the one that formed a coherent and meaningful English sentence, as the correct shift would not produce gibberish.

2.3. Execution and Results

The brute-force attack quickly generated 25 potential plaintexts. Upon manual inspection, most outputs were nonsensical. However, one shift value produced a perfectly readable English phrase.

Key (Shift): 10
Decrypted Plaintext:`ethereum is the best smart contract platform out there`

Adding spaces for readability, this becomes: “ethereum is the best smart contract platform out there”
This result was immediately identifiable as the correct decryption without any need for further manipulation.

2.4. Conclusion for Checkpoint 1

The Caesar cipher was broken effortlessly. Its primary weakness is the extremely small key space, which allows a brute-force attack to be completed in milliseconds. A modern computer can exhaust all possibilities almost instantaneously, making this cipher completely insecure for any practical use.

3. Checkpoint 2: Breaking the Substitution Cipher

3.1. Ciphertexts

1. Cipher-1: A long, single-case paragraph.
2. Cipher-2: Another long, single-case paragraph.

3.2. Approach and Methodology

A mono-alphabetic substitution cipher uses a fixed, random permutation of the alphabet to map each plaintext letter to a ciphertext letter. This creates a much larger key space (26! possibilities) than the Caesar cipher, making a pure brute-force attack computationally infeasible. Therefore, a more intelligent approach is required.

The methodology employed was a combination of **Frequency Analysis** and a **Hill-Climbing Algorithm**, followed by **Manual Intervention**.

1. Frequency Analysis: The provided frequency distribution of English characters was used as a baseline. The program began by counting the frequency of each character in the ciphertext

and mapping the most frequent ciphertext characters to the most frequent English plaintext characters (e.g., `e`, `t`, `a`, `o`, `i`, `n`).

2. Hill-Climbing Algorithm: An algorithm was implemented to refine this initial guess. It started with the frequency-based mapping and then made small, random swaps to the substitution key. If a swap increased the "fitness" of the decrypted text (i.e., the resulting text looked more like English, based on quadgram statistics), the new key was kept. This process iterated thousands of times to climb towards a local maximum of text fitness.
3. Manual Intervention: After the algorithm converged, the output was inspected. While often close to correct, it rarely produced a perfect translation. At this stage, manual deduction was crucial. We looked for:

- Partial Words: Identifying partially correct words (e.g., `th*nk` for `think`).
- Common Patterns: Recognizing common word structures and digraphs/trigraphs (e.g., `the`, `ing`, `ion`).
- Contextual Guessing: Using the context of the sentence to guess the correct letters for misplaced characters.

3.3. Execution and Results

The decryption process for the two ciphers yielded significantly different levels of difficulty.

3.3.1. Cipher-1

Process: The initial frequency analysis and hill-climbing algorithm produced a text that was already very close to perfect English. The structure of the sentences was clear, and only a handful of characters required manual correction. For instance, after the algorithm decrypted most of it, words were easily recognizable, and incorrect letters could be deduced logically from their context.
Decrypted Plaintext (Approximate): In a particular case, in each case, different ways, these moves were, based upon the quick understanding of psychohistory and the vast possibilities of its application. It was comforting to know that if anything happened to Seldon himself before the mathematics of the field were completed, there would remain one good man to continue the research.

3.3.2. Cipher-2

Process: Cipher-2 proved to be substantially more challenging. The output from the hill-climbing algorithm was significantly more garbled than that of Cipher-1. While some common words and patterns were decrypted correctly, large portions of the text remained obscure. Extensive manual substitution was required, involving trial and error for many words. The text's meaning had to be pieced together slowly, and achieving a fully meaningful and understandable paragraph was a laborious task.

Decrypted Plaintext (Approximate): bilbo was very rich and very peculiar, and had been the wonder of the shire for sixty years, ever since his remarkable disappearance and unexpected return. the riches he had brought back from his travels had nowbecome a local legend, and it was popularly believed, whatever the old folk might say, that the hill at bag endwas full of tunnels stuffed with treasure. and if that was not enough for fame, there was also his prolonged vigour to marvel at. time wore on, but it seemed to have little effect on mr. baggins. at ninety he was much the same as at fifty. at ninety-nine they began to call him well-preserved; but unchanged would have been nearer the mark. there were some that shook their heads and thought this was too much of a good thing; it seemed unfair that anyone should possess (apparently) perpetual youth as well as (reputedly) inexhaustible wealth. it will have to be paid for, they said. it isn't natural, and trouble will come of it! but so far trouble had not come; and as mr. baggins was generous with his money, most people were willing to forgive him his oddities and his good fortune. he remained on visiting terms with his relatives (except, of course, the sackville bagginses), and he had many devoted admirers among the hobbits of poor and unimportant families. but he had no close friends, until some of his younger cousins began to grow up

3.4. Comparative Analysis: Cipher-1 vs. Cipher-2

Cipher-1 was unequivocally easier to break than Cipher-2. The reasons are likely:

1. **Closer Frequency Match:** The letter frequency distribution in Cipher-1's plaintext likely matched the standard English frequency table more closely. This gave the hill-climbing algorithm a much stronger starting point, allowing it to converge on a nearly correct solution.
2. **Algorithmic Efficiency:** The fitness function of the hill-climbing algorithm worked more effectively on Cipher-1, suggesting its text structure was more "English-like."
3. **Reduced Manual Effort:** The need for manual intervention was drastically lower for Cipher-1. In contrast, breaking Cipher-2 felt like solving a complex cryptogram where the algorithm provided only a basic scaffold, and the rest had to be built manually.

Lab 3

Objective:
To perform symmetric encryption and hashing using OpenSSL and analyze the properties of different encryption modes and hash functions.

Task 1: AES Encryption Using Different Modes
Commands Used:
```bash
```

```
openssl enc -aes-128-ecb -e -in plain.txt -out cipher_ecb.bin -K
00112233445566778889aabbccddeeff
openssl enc -aes-128-cbc -e -in plain.txt -out cipher_cbc.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Decryption Test:
```bash
openssl enc -aes-128-ecb -d -in cipher_ecb.bin -out decrypted_ecb.txt -K
00112233445566778889aabbccddeeff
openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out decrypted_cbc.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypted_cfb.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observation:
All decrypted files matched the original plaintext, confirming that encryption and decryption were performed correctly.

Task 2: Encryption Mode – ECB vs CBC
Commands Used:
```bash
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bin -K
00112233445566778889aabbccddeeff
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Header Replacement:
- Copied the first 54 bytes from `pic_original.bmp` to `pic_ecb.bin` and `pic_cbc.bin` using GHex.

Observation:
ECB Mode: The encrypted image revealed visible patterns from the original image.
CBC Mode: The encrypted image appeared completely random, with no discernible patterns.
This demonstrates that ECB is insecure for encrypting data with repeating patterns, while CBC provides better security.

Task 3: Corrupted Cipher Text
Commands Used:
```bash
openssl enc -aes-128-ecb -e -in plain.txt -out cipher_ecb.bin -K
00112233445566778889aabbccddeeff
openssl enc -aes-128-cbc -e -in plain.txt -out cipher_cbc.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-ofb -e -in plain.txt -out cipher_ofb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Corruption:
- Flipped one bit at the 30th byte in each encrypted file using GHex.

Decryption After Corruption:
- ECB: Only one block (16 bytes) was affected.
- CBC: Two blocks were affected due to error propagation.
- CFB: Two blocks were affected.
- OFB: Only one byte was affected.

Explanation:
- ECB and OFB are stream-like modes, so errors are localized.
- CBC and CFB propagate errors to subsequent blocks due to chaining.

Implication:
Error propagation must be considered when choosing an encryption mode for data integrity and recovery.

Task 4: Padding
Commands Used:
```bash
openssl enc -aes-128-ecb -e -in plain.txt -out padded_ecb.bin -K
00112233445566778889aabbccddeeff

openssl enc -aes-128-cbc -e -in plain.txt -out padded_cbc.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb -e -in plain.txt -out unpadded_cfb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

openssl enc -aes-128-ofb -e -in plain.txt -out unpadded_ofb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observation:

- Padding Required: ECB, CBC
- No Padding: CFB, OFB

Explanation:
CFB and OFB are stream cipher modes and do not require padding because they encrypt data bit-by-bit or byte-by-byte.

**Task 5: Generating Message Digest**
Commands Used:
```bash
openssl dgst -md5 plain.txt
openssl dgst -sha1 plain.txt
openssl dgst -sha256 plain.txt
```

Output :
```bash
MD5(plain.txt)= f92d88d0d28d174f3b31a4a920d0a4ee
SHA1(plain.txt)= af9e0117e781752d7f7d7a497ce2da19ab479257
SHA256(plain.txt) =
368f455f99a244b80f7fcd704790bee7c560480522d5cfc404912d3ad4b94b8f
```

Observation:
Each algorithm produced a unique fixed-length hash, with SHA-256 being the longest and most secure.

Task 6: Keyed Hash and HMAC
Commands Used:
```bash
openssl dgst -md5 -hmac "mysecretkey" hmac.txt
openssl dgst -sha1 -hmac "mysecretkey" hmac.txt
openssl dgst -sha256 -hmac "mysecretkey" hmac.txt
```

Output :
```
HMAC-MD5(plain.txt)= de511705470de6efb789ec805694fc2c
```
```

HMAC-SHA1(plain.txt)=c43e7d195fb460612c4938c29b964994a653a7f3
HMAC-SHA256(plain.txt)=
3ccc72957fd4df304c66030b4563bf5dc05b09332361d2c383edad2c456a2819
```

Observation:
HMAC does not require a fixed key size. The key is hashed to match the block size of the underlying hash function.

**Task 7: One-Way Hash Properties**
Commands Used:
```bash
openssl dgst -md5 plain.txt
openssl dgst -md5 plain_modified.txt
openssl dgst -sha256 plain.txt
openssl dgst -sha256 plain_modified.txt
```

Observation:
H1 and **H2** were completely different for both MD5 and SHA-256.
- Even a single-bit change in the input resulted in a significantly different hash output.
- This demonstrates the **avalanche effect** in cryptographic hash functions.

Lab 4 – Programming Symmetric & Asymmetric Cryptography

Objective

The main objective of this lab is to understand and implement both symmetric and asymmetric cryptographic systems using Python.
Students will practically implement AES (Advanced Encryption Standard) and RSA (Rivest–Shamir–Adleman) algorithms, observe their
operations, analyze execution time performance, and explore digital signature and hashing mechanisms for data security.
```

Introduction

Cryptography is the art of securing communication and data using mathematical techniques.
In modern cryptography, algorithms are divided into two categories: symmetric and asymmetric.
- Symmetric cryptography uses a single secret key for both encryption and decryption (e.g., AES).
- Asymmetric cryptography uses two keys – public and private (e.g., RSA).

In this lab, we implement both AES and RSA cryptosystems, perform encryption/decryption, generate signatures, verify authenticity,
and measure computational performance for different key sizes.

Part A: AES (Advanced Encryption Standard)

AES is a symmetric block cipher that operates on fixed 128-bit blocks and supports key sizes of 128, 192, and 256 bits.
It is widely used due to its efficiency and strength. AES transforms plaintext into ciphertext through multiple rounds of substitution, permutation, and mixing.
Modes of operation like ECB (Electronic Codebook) and CFB (Cipher Feedback) determine how blocks are processed.

Implementation Steps for AES

1. Generate a random AES key using PyCryptodome.
2. Encrypt plaintext using AES in ECB or CFB mode.
3. Save ciphertext to a binary file.
4. Decrypt ciphertext using the same key and mode to recover plaintext.
5. Measure encryption and decryption times for different key sizes (128, 192, 256 bits).

Example Commands (in Ubuntu Terminal):

```
echo "This is AES encryption test." > message.txt
python3 crypto_lab.py aes --encrypt --mode ecb --keysize 128 --input message.txt --output aes_ecb_enc.bin
python3 crypto_lab.py aes --decrypt --mode ecb --keysize 128 --input aes_ecb_enc.bin --output aes_ecb_dec.txt
```

AES Output and Result

Plaintext: This is AES encryption test.
Ciphertext: (Binary data in aes_ecb_enc.bin)
Decrypted Text: This is AES encryption test.

AES successfully encrypted and decrypted the text using the same secret key.
Execution time increases slightly with key size but remains fast and consistent compared to RSA.

Part B: RSA (Rivest–Shamir–Adleman)

RSA is an asymmetric cryptographic algorithm that uses two keys:
- Public key for encryption or signature verification.
- Private key for decryption or signature generation.

RSA relies on the mathematical difficulty of factoring large prime numbers. It provides data confidentiality and authenticity.

Implementation Steps for RSA

1. Generate RSA key pair (private.pem and public.pem).
2. Encrypt plaintext using the public key and save ciphertext.
3. Decrypt ciphertext using the private key to recover plaintext.
4. Generate and verify digital signatures for message integrity.
5. Measure encryption/decryption times for various key sizes.

Example Commands (in Ubuntu Terminal):

```
python3 crypto_lab.py rsa --generate --keysize 2048
python3 crypto_lab.py rsa --encrypt --input message.txt --output rsa_enc.bin
python3 crypto_lab.py rsa --decrypt --input rsa_enc.bin --output rsa_dec.txt
python3 crypto_lab.py rsa --sign --input message.txt --output signature.bin
python3 crypto_lab.py rsa --verify --input message.txt --signature signature.bin
```

RSA Output and Result

Plaintext: This is RSA encryption test for Lab 4.
Ciphertext: (Binary data in rsa_enc.bin)

Decrypted Text: This is RSA encryption test for Lab 4.

Signature verification successful → ensures authenticity and integrity.
RSA execution time increases exponentially with key size due to complex modular arithmetic operations.

Execution Time Analysis

Algorithm	Key Size (bits)	Execution Time (s)
AES	128	0.0013
AES	192	0.0016
AES	256	0.0020
RSA	1024	0.0128
RSA	2048	0.0437
RSA	4096	0.1874

Observation:

- AES is efficient and consistent for all key sizes.
- RSA time increases sharply with key size.
- AES is suitable for bulk encryption; RSA is ideal for secure key exchange and digital signatures.

Conclusion

Both AES and RSA cryptography were successfully implemented using Python on Ubuntu.
AES demonstrated high speed for data encryption, while RSA showed superior security features for authentication and key management.
The lab clearly illustrates the trade-off between speed and security in symmetric and asymmetric systems.

Web Manual 1

Task 1: Apache Web Server Setup

Installation Process

Commands Executed:

```bash
sudo apt update
sudo apt install apache2
sudo ufw app list
sudo ufw allow 'Apache'
sudo ufw status
```

Service Management:

```bash
sudo systemctl status apache2
sudo systemctl start apache2
```

Local Domain Configuration:

```bash
sudo nano /etc/hosts
```

Added: `127.0.0.1 webserverlab.com`

Verification Commands:

```bash
# Test Apache status
sudo systemctl status apache2

# Test configuration
sudo apache2ctl configtest
```

Task 2: Virtual Host Configuration

Single Virtual Host Setup (example.com)

Directory Structure Creation:

```bash
sudo mkdir -p /var/www/example.com/html
sudo chown -R $USER:$USER /var/www/example.com/html
sudo chmod -R 755 /var/www/example.com
```

HTML Content Creation:
```bash
nano /var/www/example.com/html/index.html
```

Virtual Host Configuration:
```bash
sudo nano /etc/apache2/sites-available/example.com.conf
```

Virtual Host Activation:
```bash
sudo a2ensite example.com.conf
sudo a2dissite 000-default.conf
sudo apache2ctl configtest
sudo systemctl restart apache2
```

Domain Mapping:
```bash
sudo nano /etc/hosts
```
Added: `127.0.0.1 example.com`

Multiple Virtual Host Setup (anothervhost.com)

Directory Creation:
```bash
sudo mkdir -p /var/www/anothervhost.com/html
sudo chown -R $USER:$USER /var/www/anothervhost.com/html
sudo chmod -R 755 /var/www/anothervhost.com
```

HTML Content:
```bash
nano /var/www/anothervhost.com/html/index.html
```

Virtual Host Configuration:
```bash
sudo nano /etc/apache2/sites-available/anothervhost.com.conf
```

Site Activation:
```bash
sudo a2ensite anothervhost.com.conf
sudo apache2ctl configtest
sudo systemctl restart apache2
```

Domain Mapping:
```bash
sudo nano /etc/hosts
```
Added: `127.0.0.1 anothervhost.com`

Task 3: Dynamic Website Deployment

Calculator Application (example.com)

File Creation:
```bash
nano /var/www/example.com/html/calculator.html
```

Calculator HTML/JavaScript Code
```
<html>
<head>
   <title>Simple Calculator</title>
</head>
<body>
   <h1>Simple Calculator</h1>
   <form>
      First Number: <input type="number" id="num1"><br>
      Second Number: <input type="number" id="num2"><br>
      <button type="button" onclick="calculate()">Calculate</button>
   </form>
   <div id="result"></div>
   <script>
   function calculate() {
      var num1 = parseFloat(document.getElementById('num1').value);
      var num2 = parseFloat(document.getElementById('num2').value);
```

```
        var addition = num1 + num2;
        var subtraction = num1 - num2;
        var multiplication = num1 * num2;
        var division = num2 !== 0 ? num1 / num2 : "Cannot divide by zero";

        var resultHTML = "<h3>Results:</h3>";
        resultHTML += "<p>" + num1 + " + " + num2 + " = " + addition + "</p>";
        resultHTML += "<p>" + num1 + " - " + num2 + " = " + subtraction + "</p>";
        resultHTML += "<p>" + num1 + " × " + num2 + " = " + multiplication + "</p>";
        resultHTML += "<p>" + num1 + " ÷ " + num2 + " = " + division + "</p>";

        document.getElementById('result').innerHTML = resultHTML;
    }
    </script>
</body>
</html>
```

User Registration Form (anothervhost.com)

```bash
nano /var/www/anothervhost.com/html/userinfo.html
```

Registration Form HTML/JavaScript Code

```
<html>
<head>
    <title>User Information</title>
</head>
<body>
    <h1>User Information Form</h1>
    <form>
        Full Name: <input type="text" id="fullname"><br>
        Email: <input type="email" id="email"><br>
        Age: <input type="number" id="age" min="1" max="120"><br>
        <button type="button" onclick="processForm()">Submit</button>
    </form>
    <div id="output"></div>
    <script>
    function processForm() {
        var name = document.getElementById('fullname').value;
        var email = document.getElementById('email').value;
        var age = document.getElementById('age').value;
```

```
        var errors = [];
        if (name.length < 2) errors.push("Name must be at least 2 characters long");
        if (!email.includes('@') || !email.includes('.')) errors.push("Please enter a valid email
address");
        if (age < 18) errors.push("You must be at least 18 years old");

        var outputHTML = "";
        if (errors.length > 0) {
            outputHTML = "<h3 style='color: red'>Errors:</h3><ul>";
            for (var i = 0; i < errors.length; i++) {
                outputHTML += "<li>" + errors[i] + "</li>";
            }
            outputHTML += "</ul>";
        } else {
            outputHTML = "<h3 style='color: green'>Submission Successful!</h3>";
            outputHTML += "<p><strong>Name:</strong> " + name + "</p>";
            outputHTML += "<p><strong>Email:</strong> " + email + "</p>";
            outputHTML += "<p><strong>Age:</strong> " + age + " years old</p>";
        }
        document.getElementById('output').innerHTML = outputHTML;
    }
    </script>
</body>
</html>
```

Virtual Host Management Commands

Status Checking Commands:

```bash
Check enabled sites
sudo a2query -s

Check Apache status
sudo systemctl status apache2

Test configuration
sudo apache2ctl configtest
```

Site Management Commands:

```bash
 Enable sites
```

sudo a2ensite example.com.conf
sudo a2ensite anothervhost.com.conf

Disable sites
sudo a2dissite 000-default.conf
sudo a2dissite example.com.conf
sudo a2dissite anothervhost.com.conf

Restart services
sudo systemctl restart apache2
sudo systemctl reload apache2
```

**Testing Commands**

Browser Testing URLs:
```bash
Test virtual hosts
http://example.com
http://anothervhost.com

Test dynamic applications
http://example.com/calculator.html
http://anothervhost.com/userinfo.html

Test local access
http://localhost
http://127.0.0.1
http://webserverlab.com
```

Terminal Testing:
```bash
Test using curl
curl http://example.com
curl http://anothervhost.com