CPSC 526 Winter 2025
Assignment 3

Abdelrehman Abbas
Issam Akhtar

Task 1: Decryption with Password (dekrypt1.py)

Our solution for Task 1 reads encrypted data from standard input, extracts the necessary cryptographic parameters, and performs decryption using the provided password as input. The following are key code snippets that we use in our scritp.

Argument Parsing

```python
def parse_args():
    parser = argparse.ArgumentParser(
        prog='dekrypt1',
        description='AES decryptor',
    )
    parser.add_argument('password', help='password used for decryption')
    return parser.parse_args()
```

This function sets up command-line argument parsing to accept the decryption password. This function is used in all three scripts to read input arguments.

Key Derivation

```python
def key_stretch(password: str, salt: bytes, key_len: int) -> bytes:
    key = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=key_len,
        salt=salt,
        iterations=10
    ).derive(password.encode())
    return key
```

Here we derive an AES-128 key from the password and salt using PBKDF2 with 10 iterations, matching the key derivation process used in the encryption program. This function is used in all three scripts to derive the key

Decryption Process

```
def decrypt_stdin(password: str):
    raw = sys.stdin.buffer.read(32)
    if len(raw) < 32:
        return

    iv = raw[:16]
    salt = raw[16:32]

    key = key_stretch(password, salt, 16)

    decryptor = Cipher(algorithms.AES(key), modes.CTR(iv)).decryptor()

    while True:
        block = sys.stdin.buffer.read(4096)
        if not block:
            break
        pblock = decryptor.update(block)
        sys.stdout.buffer.write(pblock)

    pblock = decryptor.finalize()
    sys.stdout.buffer.write(pblock)
```

Here we read the encrypted data and extract the IV and salt. We then call the key_stretch function that derives the key, and performs AES-CTR decryption on the input data and finally writes the decrypted plaintext to standard output.

Task 2: Brute-force Password Guessing (dekrypt2.py)

Our solution for Task 2 generates potential passwords based on a pattern and checks each one against the encrypted data with support for wildcards in the form of an underscore in the provided argument password.

Password Pattern Generation

```
def regex_pass(pattern: str):
    expansions = ['']
    for ch in pattern:
        new_expansions = []
        if ch != '_':
            for e in expansions:
                new_expansions.append(e + ch)
        else:
            for e in expansions:
                for digit in '0123456789':
                    new_expansions.append(e + digit)
                new_expansions.append(e)
        expansions = new_expansions
    return expansions
```

This function generates all possible password candidates by expanding the input pattern. Each underscore in the pattern is to be replaced by any digit (0-9) or removed to create all possible password combinations.

Password Guessing and Validation

```
def guess_password(pattern: str):
    first32 = sys.stdin.buffer.read(32)
    if len(first32) < 32:
        print("no password candidates found")
        return
    iv = first32[:16]
    salt = first32[16:32]

    ciphertext = sys.stdin.buffer.read(4096)

    found_any = False
    tried = set()

    for candidate in regex_pass(pattern):
```

```
        if candidate in tried:
            continue
        tried.add(candidate)

        key = key_stretch(candidate, salt, 16)

        decryptor = Cipher(algorithms.AES(key), modes.CTR(iv)).decryptor()
        plainblock = decryptor.update(ciphertext) + decryptor.finalize()

        if plainblock.isascii():
            print(f"found password that seems to work: '{candidate}'")
            found_any = True

    if not found_any:
        print("no password candidates found")
```

This function generates password candidates using the previous function, derives keys for each, and attempts decryption. It checks if the decrypted data consists of valid ASCII characters to determine if a candidate password is potentially correct. We use a set and iterables for optimized speed of the program execution as we need to finish the program in under 85 seconds.

Task 3: Decryption Without Password (dekrypt3.py)

Our solution for Task 3 exploits the properties of CTR mode encryption when nonces are reused to recover plaintext without knowing the password.

Nonce Comparison and Plaintext Recovery

```
def dekrypt(p1file, c1file, c2file):
    with open(p1file, 'rb') as f:
        p1 = f.read()
    with open(c1file, 'rb') as f:
        c1 = f.read()
    with open(c2file, 'rb') as f:
        c2 = f.read()

    if len(c1) < 32 or len(c2) < 32:
        print("Error: cannot decrypt")
        return

    if c1[:32] != c2[:32]:
        print("Error: cannot decrypt")
```

```
        return

    c1_enc = c1[32:]
    c2_enc = c2[32:]

    n = min(len(p1), len(c1_enc), len(c2_enc))
    if n == 0:
        print("Error: cannot decrypt")
        return

    p2_recovered = bytearray(n)
    for i in range(n):
        p2_recovered[i] = c2_enc[i] ^ c1_enc[i] ^ p1[i]

    try:
        print(p2_recovered.decode('utf-8', errors='replace'), end='')
    except:
        sys.stdout.buffer.write(p2_recovered)
```

This function reads the known plaintext and both ciphertexts. It checks if the nonces (IV and salt) match, then recovers the unknown plaintext using the relationship between the ciphertexts and known plaintext through XOR operations. This works because both ciphertexts were encrypted with the same keystream (due to reused nonces).

Test Script

We have written a simple test script that goes over the test cases for each task. Run it as follows:

```
[UC issam.akhtar@csx3 assignment3] ./test.sh
Usage: ./test.sh <task_number>
Where <task_number> is 1, 2, or 3
```

Output for Task 1:

```
Test Case 1: Basic Text File Decryption
----------------------------------------
Encrypting sample1.txt...

./enkrypt.py cpsc526 < inp/sample1.txt > out/sample1.txt.enc
Decrypting sample1.txt.enc...

./dekrypt1.py cpsc526 < out/sample1.txt.enc
What is the answer to life, universe and everything?

Expected Output: What is the answer to life, universe and everything?

Test Case 2: Binary File Decryption
-------------------------------------
Encrypting tower.png...

./enkrypt.py YYC < inp/tower.png > out/tower.png.enc
Decrypting tower.png.enc...

./dekrypt1.py YYC < out/tower.png.enc > out/decrypted_tower.png
Checking SHA256 hash...

sha256sum inp/tower.png out/decrypted_tower.png
e471da531b4e9b13e6f59b99df1a005b33bf58a800b3bc87ff0cdefa1612207a  inp/tower.png
e471da531b4e9b13e6f59b99df1a005b33bf58a800b3bc87ff0cdefa1612207a  out/decrypted_tower.png

Test Case 3: Incorrect Password
---------------------------------
Decrypting sample1.txt.enc with wrong password...

./dekrypt1.py wrongpass < out/sample1.txt.enc
◆8◆◆xÔS◆◆◆6◆TX◆:◆Y◆j◆Y◆◆i◆9◆◆◆◆z◆
◆ĕ◆A◆◆
Expected Output: Garbage data (different from original)

Test Case 4: Empty File
-------------------------
Creating empty.txt...

touch empty.txt
Encrypting empty.txt...

./enkrypt.py cpsc526 < empty.txt > out/empty.txt.enc
Decrypting empty.txt.enc...

./dekrypt1.py cpsc526 < out/empty.txt.enc > out/decrypted_empty.txt
Listing decrypted file...

ls -l out/decrypted_empty.txt
-rw------- 1 issam.akhtar csusers 0 Mar 10 13:50 out/decrypted_empty.txt

Test Case 5: Large File
-------------------------
Creating large_file.bin...

dd if=/dev/urandom of=large_file.bin bs=1M count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0.0403831 s, 260 MB/s
Encrypting large_file.bin...

./enkrypt.py mypassword < large_file.bin > out/large_file.bin.enc
Decrypting large_file.bin.enc...

./dekrypt1.py mypassword < out/large_file.bin.enc > out/decrypted_large_file.bin
Checking SHA256 hash...

sha256sum large_file.bin out/decrypted_large_file.bin
160efb10043faeed98eea322b153113cccb6a6a57d5451bf94650344cd4624e2  large_file.bin
160efb10043faeed98eea322b153113cccb6a6a57d5451bf94650344cd4624e2  out/decrypted_large_file.bin

Test Case 6: Short Message
---------------------------
Encrypting short message...

echo "hello world" | ./enkrypt.py YYC > out/short_message.enc
Decrypting short_message.enc...

./dekrypt1.py YYC < out/short_message.enc
hello world
```

Output for Task 2:

```
Running Task 2 Test Cases
=========================

Test Case 1: Non-trivial password guess
---------------------------------------
Encrypting sample1.txt with password 'cpsc526'...

./enkrypt.py cpsc526 < inp/sample1.txt > out/sample1.txt.enc
Running dekrypt2.py with pattern 'cpsc52_'...

./dekrypt2.py cpsc52_ < out/sample1.txt.enc
found password that seems to work: 'cpsc526'

Test Case 2: One underscore at the end
--------------------------------------
Encrypting sample1.txt with password 'cpsc5263'...

./enkrypt.py cpsc5263 < inp/sample1.txt > out/sample1_1.txt.enc
Running dekrypt2.py with pattern 'cpsc526_'...

./dekrypt2.py cpsc526_ < out/sample1_1.txt.enc
found password that seems to work: 'cpsc5263'

Test Case 3: 2-6 underscores at the end
---------------------------------------
Testing with 2 underscores...

Encrypting sample1.txt with password 'cpsc00'...

./enkrypt.py cpsc00 < inp/sample1.txt > out/sample1_2.txt.enc
Running dekrypt2.py with pattern 'cpsc__'...

./dekrypt2.py cpsc__ < out/sample1_2.txt.enc
found password that seems to work: 'cpsc00'

Testing with 3 underscores...

Encrypting sample1.txt with password 'cpsc123'...

./enkrypt.py cpsc123 < inp/sample1.txt > out/sample1_3.txt.enc
Running dekrypt2.py with pattern 'cpsc___'...

./dekrypt2.py cpsc___ < out/sample1_3.txt.enc
found password that seems to work: 'cpsc123'

Testing with 4 underscores...

Encrypting sample1.txt with password 'cpsc4567'...

./enkrypt.py cpsc4567 < inp/sample1.txt > out/sample1_4.txt.enc
Running dekrypt2.py with pattern 'cpsc____'...

./dekrypt2.py cpsc____ < out/sample1_4.txt.enc
found password that seems to work: 'cpsc4567'

Testing with 5 underscores...

Encrypting sample1.txt with password 'cpsc89012'...

./enkrypt.py cpsc89012 < inp/sample1.txt > out/sample1_5.txt.enc
Running dekrypt2.py with pattern 'cpsc_____'...

./dekrypt2.py cpsc_____ < out/sample1_5.txt.enc
found password that seems to work: 'cpsc89012'

Testing with 6 underscores...

Encrypting sample1.txt with password 'cpsc345678'...

./enkrypt.py cpsc345678 < inp/sample1.txt > out/sample1_6.txt.enc
Running dekrypt2.py with pattern 'cpsc______'...

./dekrypt2.py cpsc______ < out/sample1_6.txt.enc
found password that seems to work: 'cpsc345678'

real    0m46.970s
user    0m46.543s
sys     0m0.138s
```

Note: Execution time for 6 wildards is under 85 seconds

Output for Task 3:

```
Running Task 3 Test Cases
=========================

Test Case 1: Successful Decryption with Reused Nonces
-----------------------------------------------------
Encrypting 1984.txt with nonce 123...

./enkrypt.py -nonce 123 unknown < inp/1984.txt > out/1984.txt.enc
Warning: non-random nonces are dangerous!
Encrypting secret.txt with nonce 123...

./enkrypt.py -nonce 123 unknown < inp/secret.txt > out/secret.txt.enc
Warning: non-random nonces are dangerous!
Running dekrypt3.py...

./dekrypt3.py inp/1984.txt out/1984.txt.enc out/secret.txt.enc > out/recovered_secret.txt
Checking if recovered file matches original...

diff inp/secret.txt out/recovered_secret.txt

Test Case 2: Decryption Failure with Different Nonces
-----------------------------------------------------
Encrypting 1984.txt with nonce 123...

./enkrypt.py -nonce 123 unknown < inp/1984.txt > out/1984.txt.enc
Warning: non-random nonces are dangerous!
Encrypting sample1.txt with nonce 456...

./enkrypt.py -nonce 456 unknown < inp/sample1.txt > out/sample1.txt.enc
Warning: non-random nonces are dangerous!
Running dekrypt3.py...

./dekrypt3.py inp/1984.txt out/1984.txt.enc out/sample1.txt.enc
Error: cannot decrypt

Test Case 3: Decryption with Binary Files
-----------------------------------------
Encrypting tower.png with nonce 789...

./enkrypt.py -nonce 789 unknown < inp/tower.png > out/tower.png.enc
Warning: non-random nonces are dangerous!
Encrypting secret.txt with nonce 789...

./enkrypt.py -nonce 789 unknown < inp/secret.txt > out/secret.txt.enc
Warning: non-random nonces are dangerous!
Running dekrypt3.py...

./dekrypt3.py inp/tower.png out/tower.png.enc out/secret.txt.enc > out/recovered_secret.txt
Checking if recovered file matches original...

diff inp/secret.txt out/recovered_secret.txt

Test Case 4: Decryption with Partial Overlap
--------------------------------------------
Creating shorter version of 1984.txt...

head -n 10 inp/1984.txt > inp/short_1984.txt
Encrypting short_1984.txt with nonce 123...

./enkrypt.py -nonce 123 unknown < inp/short_1984.txt > out/short_1984.txt.enc
Warning: non-random nonces are dangerous!
Encrypting full 1984.txt with nonce 123...

./enkrypt.py -nonce 123 unknown < inp/1984.txt > out/full_1984.txt.enc
Warning: non-random nonces are dangerous!
Running dekrypt3.py...

./dekrypt3.py inp/short_1984.txt out/short_1984.txt.enc out/full_1984.txt.enc > out/recovered_1984.txt
Checking if recovered portion matches...

diff inp/short_1984.txt out/recovered_1984.txt
```