Algorithm Recap: Summary of HBEA structure.

High-level summary:

1. We accept an email to generate a random key.
2. Store or retrieve a custom IP table and its inverse unique to each email.
3. Encrypt 32-bit blocks by applying a Feistel structure, emulating DES encryption.
4. Decrypt by applying the same operations in order, using the ciphertext to retrieve the plaintext.

Key Generation:

1. We hash a user provided email to generate a random key of 32 bytes (256 bits)
2. We apply the user's unique initial permutation to the key and then store the key as two halves, each with 16 bytes.

Message Processing:

3. We take the plaintext message of 32 bytes and apply the user's unique initial permutation on the message. We store this processed message in two halves, each with 16 bytes.

Encryption Process:

4. We take the right half of the permuted plaintext and expand it using the standard DES expansion table. We do this one 32-bit block at a time, so we expand each 32 bits (4 bytes) into 48 bits.
5. We expand the corresponding key block to the expanded message block.
6. We XOR the expanded key block and its corresponding expanded message block.
7. We pass the result through the standard DES S-boxes to go from 48 bits, back to 32 bits.
8. Lastly, we XOR the S-box output with the saved left half of the message that remained untouched during this process.
9. We repeat this process for each block of plaintext, concatenating the results until we are left with our ciphertext.

Decryption Process:

10. We expand a 32-bit block of ciphertext into 48 bits using our expansion table.
11. We expand the corresponding key block using the same expansion table.
12. XOR the expanded key block and its corresponding expanded ciphertext block.
13. Pass the result through the s-boxes to go return to 32 bit blocks.

14. Finally, XOR the s-box output with the original, untouched left half of the message block.
15. Repeat this process for each block of ciphertext, concatenating the results.
16. We apply the inverse permutation table, resulting in the original plaintext.

Benchmark Table: HBEA vs AES-256.

Creator	Me (Chase Hurwitz) **HBEA**	U.S. National Institute of Standards and Technology (NIST) **AES**
Cryptographic Logic	Based on DES (Data Encryption Standard), but customized with dynamic permutations and key generation	AES (Advanced Encryption Standard), specifically Rijndael algorithm
Block Size	32 bits (inherited from DES)	128 bits
Key Size	256 bits, derived from SHA-256(email) and broken into 8 one-byte segments	256 bits (32 bytes), used as a whole
Rounds	Only one – only has the capacity to perform a single round of encryption on 256 bits (DES uses 16 rounds)	14 rounds for AES-256
Key Schedule	Fixed permutation of hash-based segments and runtime-shuffled tables, although shuffling is done predictably.	Complex key expansion algorithm with round keys
Permutation Tables	Randomized per user (via JSON persistence); includes IP and IP^{-1}	Fixed S-boxes and permutation structure
Substitution	Uses 8 standard DES S-boxes	Uses a single 16×16 Rijndael S-box
Expansion	DES-style expansion to 48 bits for S-box input	No expansion (AES uses byte substitution + matrix operations)
Cryptographic Strength	Untested – 256-bit key is strong, but using emails to generate them is weak and predictable.	Industry standard — widely tested and secure against known practical attacks
Performance	Slower due to JSON I/O, SHA-256 + dynamic key generation	Highly optimized in hardware and software
Security Concerns	- If an attacker has access to a user's email, they can derive their key and access their IP IP-1	Secure against brute force, side-channel attacks, and chosen plaintext attacks

	- S-box reuse (from DES) is a known weakness - User-specific randomness must be cryptographically secure	when implemented properly
Flexibility	User-adaptive key system allows per-user customization of permutation tables	Standardized and uniform for all users
Use Case	Educational or niche personal encryption; experimental	Government, military, enterprise-grade data encryption
Standardization	None (experimental)	Fully standardized (FIPS 197)

Quantum Simulation: Output or screenshots.

Goal: Simulate a quantum attack using Grover's Algorithm.

Overview: Grover's Algorithm is a quantum search algorithm used to search unsorted, unstructured databases in $O(\sqrt{N})$ time. This is much more efficient than a classic algorithm that would rely on brute force in $O(N)$ time.

Our simulation: Our HBEA utilizes SHA-256 to create a key of 256 bits. In the real world, that would take Grover's algorithm 2^{128} iterations to find the key which is too big even for quantum computers, let alone a classic laptop with Qiskit. So instead, we scale down our key to 4 bits to demonstrate the proper quantum principles. With a 4-bit target, we utilize 4 qubits to represent the solution space which instead of having 2^{256}possibilities, now only has 2^4 or 16 possibilities which is a much more computationally feasible job for a laptop, while still being conceptually illustrative.

Justification: Grover's algorithm complexity scales at $O(\sqrt{N})$ regardless of N. Therefore, we can extrapolate our findings from a small simulation to a full 256-bit key target in order to analyze how Grover's algorithm would perform on our HBEA.

Methodology:

1. Initialization: we create a quantum circuit with 4 qubits and 4 matching classical bits.
2. Superposition: Hadamard gates are applied to all qubits, placing them in a uniform superposition of all 16 possible solutions.
3. Oracle Construction: We marked the target key |1011> by flipping any qubit where the target bit was 0. Then, we apply multi-controlled-Z operations, and then unflip those qubits which inverts the amplitude of the target key.
4. Diffusion Operator: We apply another round of Hadamard, X, and multi-controlled-Z gates to reflect the amplitudes over the mean.
5. Measurement and Simulation: We added measurement gates to all qubits and simulated the circuit 1024 times (an appropriately large number to see meaningful trends).

Results:

The measured results show our target key of |1011> occurred 455 out of 1024 times, significantly higher than any other state. All other states stayed consistently low at 27-47 measurements. This outcome demonstrates a successful implementation of Grover's Algorithm, as it increased the amplification of the correct state, while minimizing the other state's amplitudes. Therefore, these results demonstrate how Grover's Algorithm can drastically improve search results in unsorted, unstructured solution spaces.

Visualizations:

Before simulation/measurement

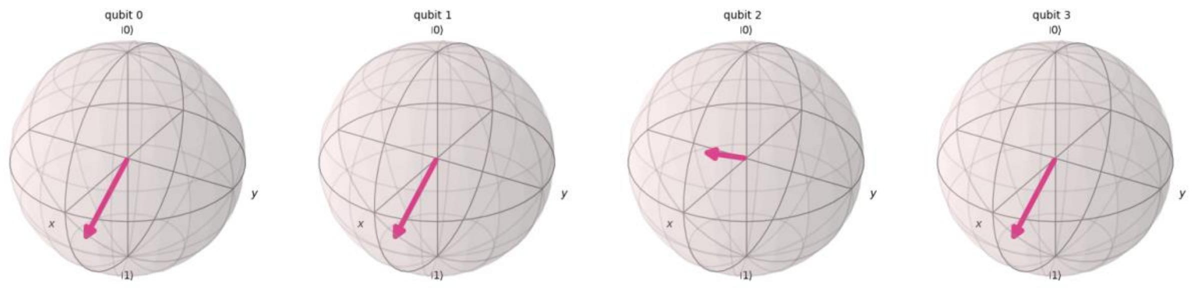


Figure 1: Bloch Spheres showing the initial state of the 4 qubits before simulation.

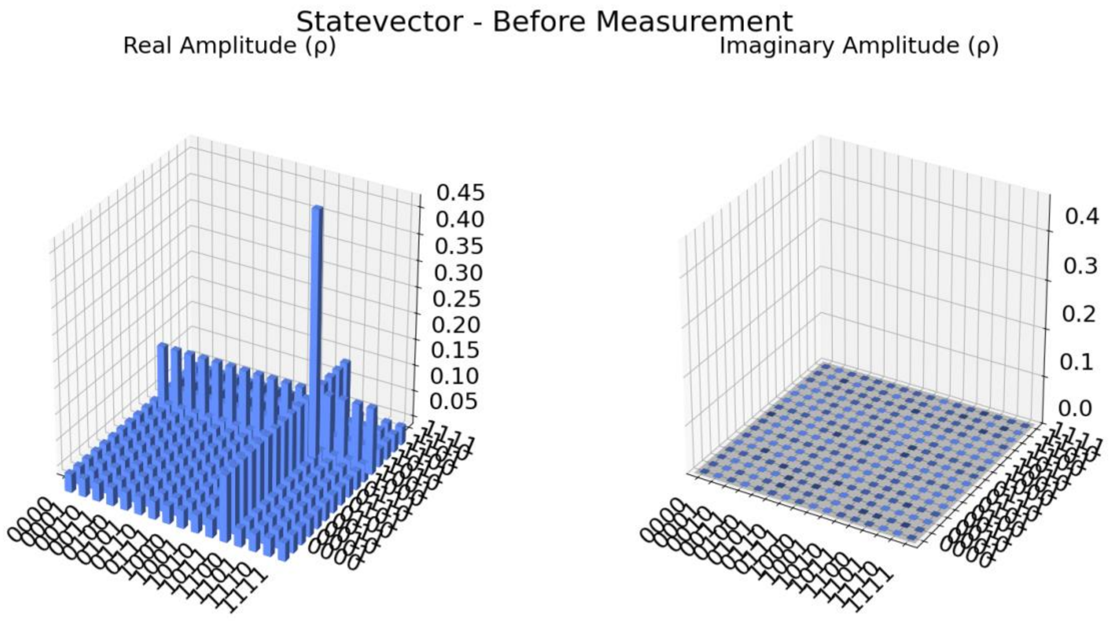
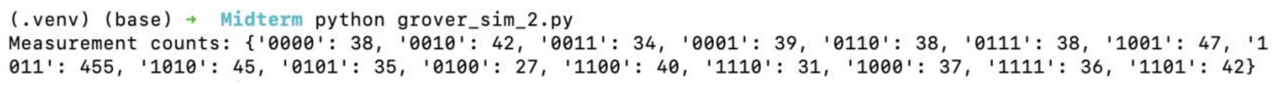


Figure 2: A 3D representation of a quantum statevector, illustrating the probability amplitudes prior to measurement, with heightened bars indicating states with larger amplitudes, and thus a higher probability of being measured.

After simulation/measurement:



```
(.venv) (base) ➜  Midterm python grover_sim_2.py
Measurement counts: {'0000': 38, '0010': 42, '0011': 34, '0001': 39, '0110': 38, '0111': 38, '1001': 47, '1
011': 455, '1010': 45, '0101': 35, '0100': 27, '1100': 40, '1110': 31, '1000': 37, '1111': 36, '1101': 42}
```

Figure 3: demonstrates the frequency each possible state was observed after running our circuit 1,024 times.

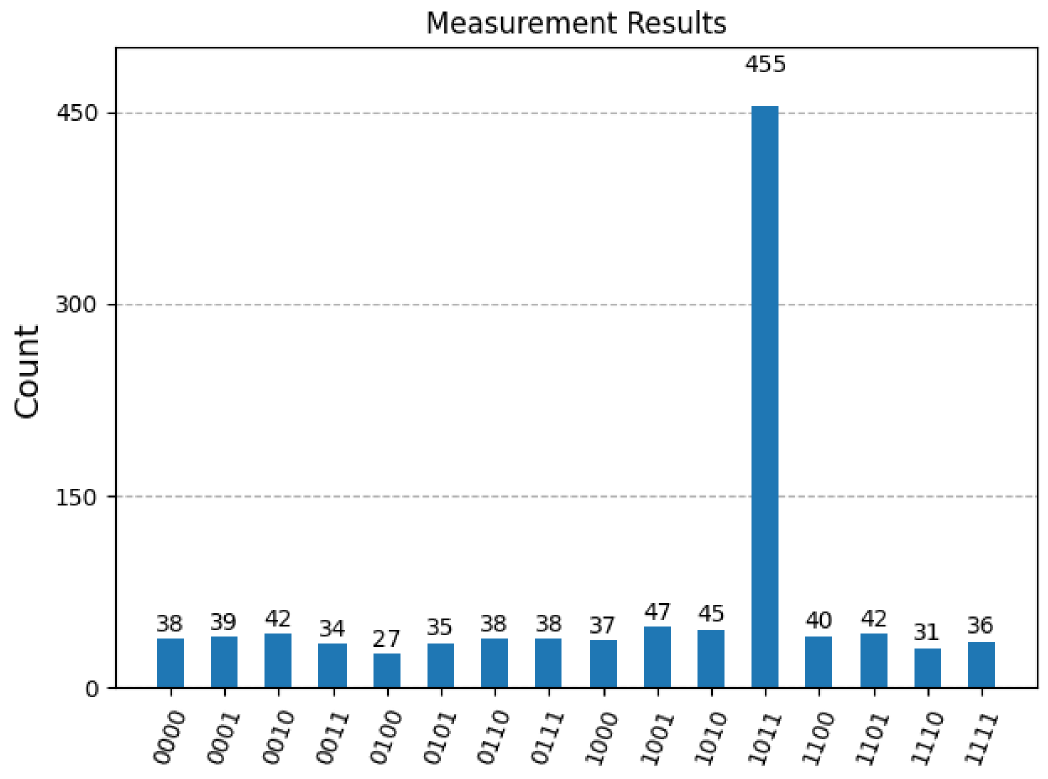


Figure 4: visualizes the measurement data in a histogram, clearly demonstrating Grover's Algorithm accurately observed our target key significantly more than others.

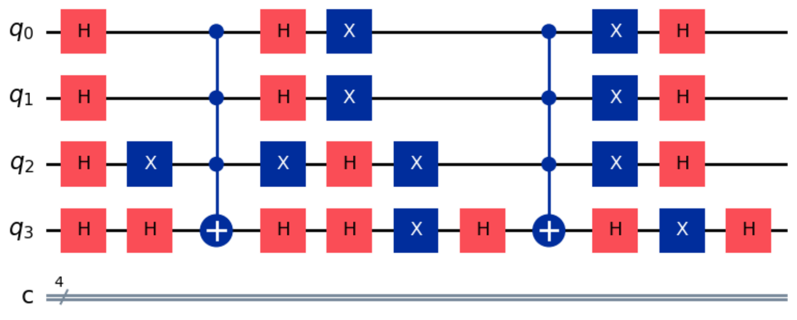


Figure 5: Grover circuit observed after measurement, showing depicting the gates.

Extrapolation:

My HBEA algorithm utilizes a 256-bit key by hashing a user's email using SHA-256. To represent this solution space, Grover's Algorithm would require 256 qubits (one per bit) to represent the superposition of all 2^{256} solutions. Additionally, we will likely need extra ancillary qubits to help perform intermediary steps like multi-controlled NOTs or phase flips. These qubits do not represent the input/output but are needed to perform complex circuit computations. In our small-sized simulation with four qubits, we did not need any ancillary qubits because the operations remained simple enough, but cracking a 256-bit key could require hundreds or thousands of ancillary qubits.

The number of Grover iterations to identify a key of length N grows at $O(\sqrt{N})$. Thus, for both my HBEA or AES-256 where N = 256, the worst-case time complexity equals 2^{128} operations, each requiring multiple quantum gate operations. Executing 2^{128} iterations is practically infeasible even for a quantum computer at this point. Thus, HBEA's strength is maintained because 2^{256} is still a sufficiently large solution space that even a quantum computer would struggle to brute force it. But, in the future, as quantum technology gets better and cheaper, and companies build quantum computers with more and more qubits, brute forcing a solution space of 2^{256} could be possible.

Quantum Resistance:

Grover's algorithm provides a quadratic speed-up over traditional brute force search. What would take $O(N)$ time, can be completed in $O(\sqrt{N})$. However, my HBEA generates its random key from SHA-256, meaning N = 256 bits, and thus Grover's Algorithm's worst-case complexity for cracking a password is 2^{128} operations. This is still sufficiently large enough to withstand brute force attacks from quantum computers. For comparison, AES-128 is still considered quantum resistant according to NIST PQC evaluation criteria. This means that Grover's Algorithm would need to complete 2^{64} operations to find the key which is still infeasible for a quantum computer. Therefore, our HBEA algorithm is quantum resistant to modern quantum attacks for the foreseeable future.

AES-256 VS HBEA:

Metric	**AES-256**	**HBEA**
Key size	256 bits	256 bits
Classic brute force operations	2^{256}	2^{256}
Grover Algorithm operations	2^{128}	2^{128}
Structure	Block cipher	Hash-based (SHA-256)
Oracle Complexity	Requires reversible AES	Requires reversible SHA-256

Theoretically, Grover's Algorithm equally weakens both AES-256 and my HBEA by reducing the solution space from 2^{256} to 2^{128}. But in practice, my HBEA may be even more secure than AES-256 due to the oracle complexity. Quantum circuits must be reversible at the gate-level: AES-256 is built with relatively simple operations (XOR, substitutions, etc.) which can be reversible. Thus, to build an AES oracle would be very complex, but possible (in fact, AES Grover Oracles already exist). But, SHA-256 is hash function and is thus intentionally built to be one-way using complex operations like bit-mixing, additions, and non-linear compressions. So, to implement the reversible circuit needed to run Grover's Algorithm would be incredibly difficult for SHA-256. Therefore, in actuality, building a SHA-256 Grover Oracle would likely be much more difficult (and require many more qubits) than building that for AES-256. Therefore, while Grover's Algorithm theoretically weakens both AES-256 and my HBEA the same amount, in practice, I believe my HBEA is actually stronger due to the one-way nature of SHA-256.

HBEA NIST Compliance Evaluation

1. Key Length: Key lengths of 256 bits are considered secure against classical brute force and quantum attacks per NIST. My HBEA meets this standard with a 256-bit key.
2. Permutation Strength: NIST recommends that block ciphers should utilize non-linear operations and diffusion to ensure secure encryption. My HBEA utilizes both of those in a Feistel structure with S-boxes, permutations, and shifting. Thus, my HBEA meets this standard.
3. Padding Scheme: NIST recommends unambiguous and securely reversible padding for block alignment. My HBEA does not have a padding scheme and requires exactly 32 characters or a full 256-bit block as plaintext. This represents a potential weakness as attackers always know the length of the plaintext. Thus, this aspect of my HBEA is not NIST compliant.
4. Random and Secure Key Generation: We use SHA-256 to generate our keys which is endorsed by NIST. That being said, we are deriving these keys from potentially predictable inputs (like emails) – so if an attacker gains access to a user's email, then they can almost immediately derive their key, meaning the secure length of 256-bits would be irrelevant. To truly be secure, we could replace SHA-256 with a key derivation function (KDF) and store each salted version in a database, keeping the functionality where each user gets a unique key, but removing the predictable email input.
5. Key Storage: keys should be stored securely with hardware protection, or encrypted software key stores, and never in plaintext. My HBEA never stores any keys directly which is good, but it does store the emails used to generate these keys in plaintext in

a JSON file. This JSON file then becomes an attack target because access to it essentially gives away all the sensitive information about a user and allows them to immediately derive their key by hashing their email. To meet NIST recommendations, it would be better to utilize a KDF with email + password + user specific salt as inputs.

Vulnerability Analysis: Issues and proposed fixes.

1. Weak key derivation:
    a. Issue: Creating a key based on a user's email alone is not secure. An attacker would only need to guess a user's email to learn their key which would be far less difficult than brute-force guessing the 256-bit key. Also, it is very likely an attacker could gain access to their email somehow which would also be insecure for our user.
    b. Solution: replace our SHA-256 key generation with an actual KDF like Argon2id.
2. Poor Data Storage:
    a. Issue: We store the user's email and their unique IP IP-1 tables in a JSON file. Incredibly sensitive information is stored in plaintext in this file and an attacker would be able to decrypt every secure message if they gained access to this JSON file.
    b. Solution: We should use more secure storage methods like a relational database with encrypted fields. If not, we could at least encrypt a user's email and IP, IP-1 tables before inserting them into the JSON file so that an attacker cannot easily gain this information if the JSON gets compromised.
3. Non-existent Data Padding:
    a. My HBEA does not implement data padding for a user's plaintext. If they do not provide a plaintext message of exactly 32 bytes, the algorithm will not accept it and instead use a default message of 32 characters. If an attacker knows that the plaintext needs to be exactly 32 characters, they can easily target that exact input length to brute force.
    b. Solution: We should utilize standard padding schemes like PKCS #7. This would add bytes to fit the necessary block size and thus add randomness to the encryption.

Source Code: Python scripts used for tests.

grover_sim_2.py:

```
'''
    -Grover Simulation
    -Author: Chase hurwitz
    -This code is created to simulate Grover's Algorithm with 4 qubits, trying to guess a target state of 4 bits.


1. Initialization: we create a quantum circuit with 4 qubits and 4 matching classical bits.
2.  Superposition: Hadamard gates are applied to all qubits, placing them in a uniform superposition of all 16 possible solutions.
3.  Oracle Construction: We marked the target key |1011> by flipping any qubit where the target bit was 0. Then, we apply multi-controlled-Z operations, and then unflip those qubits which inverts the amplitude of the target key.
4.  Diffusion Operator: We apply another round of Hadamard, X, and multi-controlled-Z gates to reflect the amplitudes over the mean.
5.  Measurement and Simulation: We added measurement gates to all qubits and simulated the circuit 1024 times (an appropriately large number to see meaningful trends).


'''



from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from qiskit.quantum_info import Statevector
import numpy as np
import matplotlib.pyplot as plt
```

```
# Parameters

n = 4


## The targest state |1011> is represented as [1,1,0,1]

## because Qiskit indexes left to right aka, Qiskit

## recognizes Big Endian format where the least significant

## byte is stored at the lowest memory address. So, [1,1,0,1] = "1011"


target = [1, 1, 0, 1]  # The target state |1011> (see note above)


# Setup

qr = QuantumRegister(n, 'q')

cr = ClassicalRegister(n, 'c')

grover_circuit = QuantumCircuit(qr, cr)


# --- Step 1: Apply superposition ---

grover_circuit.h(qr)


# --- Step 2: Oracle for |1011> ---

# Flip qubits that are 0 in the target (i.e., q[1])

for i, bit in enumerate(target):

   if bit == 0:

      grover_circuit.x(qr[i])
```

```
# Apply multi-controlled-Z (by conjugating H and using MCX)
grover_circuit.h(qr[3])
grover_circuit.mcx(qr[0:3], qr[3])  # Control on q[0], q[1], q[2]
grover_circuit.h(qr[3])

# Unflip the X gates
for i, bit in enumerate(target):
    if bit == 0:
        grover_circuit.x(qr[i])

# --- Step 3: Diffusion operator ---
grover_circuit.h(qr)
grover_circuit.x(qr)

# Apply multi-controlled-Z
grover_circuit.h(qr[3])
grover_circuit.mcx(qr[0:3], qr[3])
grover_circuit.h(qr[3])

grover_circuit.x(qr)
grover_circuit.h(qr)

# --- Visualize Statevector BEFORE measurement ---
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_bloch_multivector, plot_state_city
```

```
# Get statevector before measurement
state = Statevector.from_instruction(grover_circuit)


# Plot Bloch spheres
plot_bloch_multivector(state)
plt.savefig("bloch_multivector.png")
plt.close()


# Plot state city (real/imag amplitude bar plots)
plot_state_city(state, title="Statevector - Before Measurement")
plt.savefig("state_city.png")
plt.close()




# --- Step 4: Measure ---
grover_circuit.draw(output='mpl')
plt.savefig("grover_circuit.png")
plt.close()
grover_circuit.measure(qr, cr)


# --- Run simulation ---
sim = AerSimulator()
result = sim.run(grover_circuit, shots=1024).result()
counts = result.get_counts()


print("Measurement counts:", counts)
```

```
plot_histogram(counts, title="Measurement Results")

plt.savefig("grover_histogram.png")

plt.close()
```

Quantum Security Analysis of HBEA Final Project – Hurwitz

DES_Algo_2.py:

```
'''
    -DES Encryption/Decryption
    -Author: Chase hurwitz
    -This code accepts an email, uses it to generate a random key, and then checks if that email already has existing IP, IP inverse table by lookup in a simple
    JSON file. If not, the program generates a random IP, IP inverse table and stores that user and their tables as a dictionary in the JSON file. Next, we run
    DES Encryption on their provided plaintext (must be 32 bytes/characters) and then decrypt the CT to demonstrate successful decryption as well.
'''


import hashlib
import random
import time
import json
import os


# =================
# Global variables
# =================
PERM_DB_FILE = "perm_tables.json" #for data persistence


DES_EXPANSION_TABLE = [
    31, 0, 1, 2, 3, 4,
    3, 4, 5, 6, 7, 8,
    7, 8, 9, 10, 11, 12,
    11, 12, 13, 14, 15, 16,

    15, 16, 17, 18, 19, 20,

    19, 20, 21, 22, 23, 24,

    23, 24, 25, 26, 27, 28,

    27, 28, 29, 30, 31, 0]


# S boxes

S = [

        #S1

        [

            [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

            [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

            [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

            [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]

        ],


        #S2

        [

            [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

            [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

            [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

            [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],

        ],


        #S3

        [
            [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
            [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
            [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
            [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],
        ],

        #S4
        [
            [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
            [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
            [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
            [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],
        ],

        #S5
        [
            [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
            [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
            [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
            [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],
        ],

        #S6
        [
            [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
            [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
            [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
            [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],
        ],

        #S7
        [
            [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
            [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
            [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
            [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],
        ],

        #S8
        [
            [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
            [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
            [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
            [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11],
        ]
    ]

# ========================
# 1) KEY GENERATION
# ========================

def key_generation(email, perm_table):
```

```
# 1 hash the email to get 256 bits (32 bytes)

hash_bytes = hashlib.sha256(email.encode('utf-8')).digest()  # 32 bytes


# 2 convert the 32 bytes into a flat list of 256 bits

key_bits = bytes_to_bits(hash_bytes)


# 3 split the 256 bits into 8 segments (each 32 bits)

segments = []

for i in range(8):

    segments.append(key_bits[i*32:(i+1)*32])


# 4 concatenate the segments into 4 blocks of 64 bits using the specified indices:

block1 = segments[0] + segments[2]

block2 = segments[1] + segments[3]

block3 = segments[4] + segments[6]

block4 = segments[5] + segments[7]


#5 apply the permutation function here on each of these blocks:

permuted_block1 = apply_permutation(block1, perm_table)

permuted_block2 = apply_permutation(block2, perm_table)

permuted_block3 = apply_permutation(block3, perm_table)

permuted_block4 = apply_permutation(block4, perm_table)


#6 combine into 8 segment blocks

key_blocks = []

for block in [permuted_block1, permuted_block2, permuted_block3, permuted_block4]:
        # Each block is 64 bits long, so we spplit each in half
        first_half = block[:32]
        second_half = block[32:]
        key_blocks.append(first_half)
        key_blocks.append(second_half)

    return key_blocks


# ========================
# 2) MESSAGE PROCESSING:
#  - PERMUTATION TABLES
# ========================
def generate_perm_unperm_tables():
    indices = list (range(64))
    random.shuffle(indices)
    IP = indices[:] #take a slice

    # calculate the inverse
    IP_inv = [0]*64 #empty list of 64 zeros
    for i, val in enumerate(IP):
        IP_inv[val] = i #swap index and value back to original

    return IP, IP_inv



def apply_permutation(bit_list, perm_table):
    """
    perm_table must be the same length as bit_list (64 bits)
    """
    result = []
    for i in range(len(perm_table)):
        result.append(bit_list[perm_table[i]])
    return result


def process_message(message, perm_table):
        # 1. Convert the message (assumed 32 characters = 256 bits) to bytes and then to a flat
list of bits.
    message_bytes = message.encode("utf-8")
    message_bits = bytes_to_bits(message_bytes)


    # 2. Split the 256 bits into 8 segments (each 32 bits)
    segments = []
    for i in range(8):
        segment = message_bits[i*32:(i+1)*32]
        segments.append(segment)


    block1 = segments[0] + segments[2] #make 4 blocks according to diagram
    block2 = segments[1] + segments[3]
    block3 = segments[4] + segments[6]
    block4 = segments[5] + segments[7]


    # 4. apply the initial permutation on each 64-bit block using the provided perm_table.
    permuted_block1 = apply_permutation(block1, perm_table)
    permuted_block2 = apply_permutation(block2, perm_table)
    permuted_block3 = apply_permutation(block3, perm_table)
    permuted_block4 = apply_permutation(block4, perm_table)

    # 5. breaak each permuted 64-bit block into two 32-bit blocks, creating a total of 8
segments.
    message_segments = []
    for block in [permuted_block1, permuted_block2, permuted_block3, permuted_block4]:
        first_half = block[:32]
        second_half = block[32:]
        message_segments.append(first_half)
        message_segments.append(second_half)


    # 7. Return the list of 8 segments.
    return message_segments

# ========================
# 3) ENCRYPTION PROCESS:
#  - bits to bytes and vice versa
#  - XOR function
#  - Expansion Function
#  - S-boxes
#  - encyrption
# ========================
def bytes_to_bits(byte_data):
    bit_list = []
    for byte in byte_data:
        bits = bin(byte)[2:].zfill(8)  # Convert to binary, remove "0b", pad with zeros
        for bit in bits:
            currBit = int(bit)
            bit_list.append(currBit)
    return bit_list


def bits_to_bytes(bit_data):
    """
    Assumes len(bit_list) is a multiple of 8.
    """
    byte_list = bytearray()
    for i in range(0, len(bit_data), 8):
        byte = 0
        for j in range(8):
            byte = (byte << 1) | bit_data[i + j]
        byte_list.append(byte)
    return byte_list


def xor_bits(a, b):
    result = []
    for i in range(len(a)):
        if a[i] == b[i]:
            result.append(0)  # If bits are the same, XOR is 0
        else:
            result.append(1)  # If bits are different, XOR is 1
    return result


def expansion_32_to_48(bits):
    ''' assumes that bits is 32 bits long (indeces 0,31 inclusive)'''


    expanded_bits = []
    for i in DES_EXPANSION_TABLE:
        expanded_bits.append(bits[i])  # Append the bit at the specified index
    print("Expanded_Bits now at length 48: ", expanded_bits)
    return expanded_bits


def sbox_application(bits):
    ''' assumes that bits is 48 bits long (indeces 0,47 inclusive)'''


    output_32_bits = []
    for i in range(8):
        input_6_bits = bits[i*6:(i+1)*6]


        row = (input_6_bits[0] << 1) | input_6_bits[5]
        col = (input_6_bits[1] << 3) | (input_6_bits[2] << 2) | (input_6_bits[3] << 1) |
input_6_bits[4]


        sbox_val = S[i][row][col]
        # Convert sbox_val into 4 bits
        val_bits = [
            (sbox_val >> 3) & 1,
            (sbox_val >> 2) & 1,
            (sbox_val >> 1) & 1,
            sbox_val & 1,
        ]
        output_32_bits.extend(val_bits)
    print("After going through sbox (now 32 bits): ", output_32_bits)
    return output_32_bits


def encryption_feistel(message_segments, key_blocks):
    """
    Given message_segments and key_blocks, each a list of 8 segments (32 bits each),
encrypt in the order of
    1. Expansion
    2. XOR
    3. sbox
    4. concatentate
    """
    ciphertext = []


    # There are 4 pairs: indices: [0,1], [2,3], [4,5], [6,7]
    for i in range(4):
        left  = message_segments[2 * i]
        right = message_segments[2 * i + 1]
        # Expand the right half from 32 to 48 bits

        expanded_right = expansion_32_to_48(right)


        # use the key block corresponding to the right halvr's index

        key_block = key_blocks[2 * i + 1]


        # Expand the key  from 32 to 48 bits

        expanded_key = expansion_32_to_48(key_block)


        # XOR the expanded right with the expanded key.

        xor_result = xor_bits(expanded_right, expanded_key)


        # Pass the result through the S-box transformation (48 bits -> 32 bits

        sbox_out = sbox_application(xor_result)


        # XOR the S-box output with the saved left half

        new_left = xor_bits(left, sbox_out)


        # Form the ciphertext pair by concatenating new_left and the original right

        pair_ciphertext = right + new_left #we swap at the end to make it easier to pass through
for decryption tho


        ciphertext.extend(pair_ciphertext)


    return ciphertext
# ========================
#
# 4) DECRYPTION PROCESS:
#
# ========================

def decryption_feistel(ciphertext_bits, key_blocks, perm_inv):
    """
    Decrypt a ciphertext bascically the same we encrypt PT
    """
    recovered_plaintext = []

    for i in range(4):
        # Extract the 64-bit ciphertext pair.
        pair = ciphertext_bits[i * 64 : (i + 1) * 64]
        R = pair[:32]
        Lprime = pair[32:]

        # Expand R to 48 bits.
        expanded_R = expansion_32_to_48(R) #we use R instaed of L here because to form our
CT, we swapped the pair at the veyr end

        # Use the key block corresponding to the right half of the pair
        key_block = key_blocks[2 * i + 1]
        expanded_key = expansion_32_to_48(key_block)
        # XOR the expanded R with the expanded key

        xor_result = xor_bits(expanded_R, expanded_key)


        # Pass the result through the S-box transformation

        sbox_out = sbox_application(xor_result)


        # Recover the original left half: L = Lprime XOR sbox_out

        recovered_L = xor_bits(Lprime, sbox_out)


        # Reassemble the plaintext pair in the original order: (L, R)

        recovered_L_and_R = recovered_L + R

        permuted_concatenation_PT = apply_permutation(recovered_L_and_R, perm_inv)

        recovered_plaintext.extend(permuted_concatenation_PT)


        # Split into 8 segments of 32 bits

    segments = [recovered_plaintext[i*32:(i+1)*32] for i in range(8)]


    # Define the desired new order:

    order = [0, 2, 1, 3, 4, 6, 5, 7]


    # Reassemble the segments in the new order that matches how we originally paired the
segments

    reorganized = []

    for index in order:

        reorganized.extend(segments[index])
    return reorganized



# ========================
#
# 5) GET USER INPUT:
#
# ========================

def get_user_input():
    print("\n Welcome to DES Encryption/Decryption. Created by Chase Hurwitz")

    # Prompt for user email.
    email = input("Enter your email: ").strip()
    if not email:
        print("default email used because there was an error with yours: john.doe@IES.org")
        email = "john.doe@IES.org"

    # Prompt for a 32-character plaintext message.
    plaintext_str = input("Enter a 32-character plaintext message: ").strip()
    if len(plaintext_str) != 32:
        default_text = "0123456789ABCDEF0123456789ABCDEF"
        print("Message must be exactly 32 characters. Using default:")
        print(default_text)
        plaintext_str = default_text
    return email, plaintext_str



# =====================================================
#
# 6) IMPLEMENT DATA PERSISTENCY WITH A SIMPLE JSON FILE:
#
# =====================================================
def load_perm_tables():
    #Load the dictionary {email: {"IP": [...], "IP_inv": [...]}} from a JSON file
    if os.path.exists(PERM_DB_FILE):
        with open(PERM_DB_FILE, "r") as f:
            return json.load(f)
    else:
        return {} #make it


def save_perm_tables(data):
    #Save the permutation tables dictionary to a JSON file
    with open(PERM_DB_FILE, "w") as f:
        json.dump(data, f)
        print(f"Saved permutation tables to {PERM_DB_FILE}.") #debuger


def get_or_create_perm_tables_for_user(email):
    data = load_perm_tables()
    if email in data:  # Reuse existing IP and IP_inv
        IP_table = data[email]["IP"]
        IP_inv = data[email]["IP_inv"]

    else:  # Generate new IP_table, IP_inv for this user
        IP_table, IP_inv = generate_perm_unperm_tables()
        data[email] = {
            "IP": IP_table,
            "IP_inv": IP_inv
        }

        save_perm_tables(data)# Save to JSON file
    return IP_table, IP_inv



# =====================================================
#
# 7) MAIN FUNCTION RUNS OUR CODE
#
# =====================================================

def main():

    #1 get user input
    email, message = get_user_input()
```

```
print('\n Message:  ', message )


#2 Generate a permutation table for 64-bit blocks
perm_table, perm_inv = get_or_create_perm_tables_for_user(email)


#3 Process the plaintext message into 8 segments (32 bits each)
message_segments = process_message(message, perm_table)
print("\nProcessed Message Segments:")
for i, seg in enumerate(message_segments):
    print(f"Segment {i}: {seg}")


#4 Generate key blocks from the email using the same permutation table
key_blocks = key_generation(email, perm_table)
print("\nKey Blocks:")
for i, block in enumerate(key_blocks):
    print(f"Key Block {i}: {block}")


#5 encrypt the message using our Feistel-style encryption function
ciphertext_bits = encryption_feistel(message_segments, key_blocks)


ciphertext_bytes = bits_to_bytes(ciphertext_bits)




print("\n ============ ENCRYPTING ============")
time.sleep(1.5)
```

```
    print("\nCiphertext (hex):")
    print(ciphertext_bytes.hex())

    print("\n ============ DECRYPTING =============")
    time.sleep(1.5)

    # 6 Decrypt the ciphertext
    recovered_bits = decryption_feistel(ciphertext_bits, key_blocks, perm_inv)



    recovered_bytes = bits_to_bytes(recovered_bits)

    try:
        recovered_text = recovered_bytes.decode("utf-8")
    except UnicodeDecodeError:
        recovered_text = repr(recovered_bytes)

    print("\nRecovered Plaintext:")
    print(recovered_text)

    if recovered_text.strip() == message:
        print("\nSUCCESS: Decrypted text matches the original!")
    else:
        print("\nERROR: Decrypted text does not match the original!")
```

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

Slide Summary: 3-5 slides with brief explanation for each algorithm and highlighting your main findings, references and tools.