```python
#AES

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
import base64

def aes_encrypt(plaintext, key, mode, iv=None):
    cipher = AES.new(key, mode, iv=iv)
    ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))
    return base64.b64encode(ciphertext).decode('utf-8')

def aes_decrypt(ciphertext, key, mode, iv=None):
    cipher = AES.new(key, mode, iv=iv)
    decrypted_data = cipher.decrypt(base64.b64decode(ciphertext))
    return unpad(decrypted_data, AES.block_size)

# User Input
mode = input("Enter AES mode (ECB/CBC): ")
operation = input("Enter operation (Encrypt/Decrypt): ")
data = input("Enter plaintext/ciphertext: ")
key = input("Enter AES key (16/24/32 bytes): ")

# Validate the inputs

# Convert the key to bytes
key = key.encode('utf-8')

# Check if the mode is valid
if mode not in ['ECB', 'CBC']:
    print("Invalid AES mode.")
    exit()

# Check if the operation is valid
if operation not in ['Encrypt', 'Decrypt']:
    print("Invalid operation.")
    exit()

# Perform AES encryption or decryption based on user input
if operation == 'Encrypt':
```

```
    plaintext = data.encode('utf-8')
    if mode == 'ECB':
        encrypted_data = aes_encrypt(plaintext, key, AES.MODE_ECB)
    else:
        iv = get_random_bytes(AES.block_size)
        encrypted_data = aes_encrypt(plaintext, key, AES.MODE_CBC, iv)
        print("IV:", base64.b64encode(iv).decode('utf-8'))
    print("Ciphertext:", encrypted_data)
else:
    if mode == 'ECB':
        decrypted_data = aes_decrypt(data, key, AES.MODE_ECB)
    else:
        iv = input("Enter initialization vector (IV) in Base64 format: ")
        iv = base64.b64decode(iv)
        decrypted_data = aes_decrypt(data, key, AES.MODE_CBC, iv)
    print("Plaintext:", decrypted_data.decode('utf-8'))







#RSA

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

from Crypto.PublicKey import RSA

# Generate RSA key pair
key = RSA.generate(1024)

# Print the public key
public_key = key.publickey().export_key().decode()
print("Public Key:\n", public_key)

# Print the private key
private_key = key.export_key().decode()
print("Private Key:\n", private_key)
```

```python
def encrypt(plaintext, public_key):
    cipher_rsa = PKCS1_OAEP.new(public_key)
    ciphertext = cipher_rsa.encrypt(plaintext.encode())
    return ciphertext.hex()

def decrypt(ciphertext, private_key):
    cipher_rsa = PKCS1_OAEP.new(private_key)
    decrypted_message = cipher_rsa.decrypt(bytes.fromhex(ciphertext))
    return decrypted_message.decode()

# Example usage
operation = input("Enter operation (Encrypt/Decrypt): ").lower()
plaintext_ciphertext = input("Enter the plaintext or ciphertext: ")
key = public_key

try:
    if operation == "encrypt":
        public_key = RSA.import_key(key)
        encrypted_text = encrypt(plaintext_ciphertext, public_key)
        print("Encrypted text:", encrypted_text)
    elif operation == "decrypt":
        private_key = RSA.import_key(private_key)
        decrypted_text = decrypt(plaintext_ciphertext, private_key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid operation. Please choose 'Encrypt' or 'Decrypt'.")
except Exception as e:
    print("An error occurred:", str(e))



#Hash

import hashlib

def calculate_hash(plaintext, hash_mode):
    try:
        if hash_mode == 'SHA1':
            hash_object = hashlib.sha1()
```

```
        elif hash_mode == 'SHA256':
            hash_object = hashlib.sha256()
        else:
            return 'Invalid hash mode'

        hash_object.update(plaintext.encode('utf-8'))
        hash_value = hash_object.hexdigest()
        return hash_value
    except Exception as e:
        return str(e)

# Example usage
plaintext = input("Enter the plaintext: ")
hash_mode = input("Enter the hash mode (SHA1 / SHA256): ")

hash_value = calculate_hash(plaintext, hash_mode)
print("Hash value:", hash_value)



#Digital

import hashlib

from Crypto.PublicKey import RSA

from Crypto.Signature import pkcs1_15

from Crypto.Hash import SHA256



def generate_rsa_key_pair():

   # Generate a new RSA key pair

   key = RSA.generate(2048)

   return key
```

```
def sign_message(message, private_key):
    # Generate a hash of the message
    hash_value = SHA256.new(message.encode())

    # Create a signature using the private key
    signer = pkcs1_15.new(private_key)
    signature = signer.sign(hash_value)

    return signature


def verify_signature(message, signature, public_key):
    # Generate a hash of the message
    hash_value = SHA256.new(message.encode())

    # Verify the signature using the public key
    verifier = pkcs1_15.new(public_key)

    try:
        verifier.verify(hash_value, signature)
        return True
    except (ValueError, TypeError):
```

```python
        return False


# Example usage

operation = input("Enter the operation (Generation/Verification): ")


if operation.lower() == "generation":

   message = input("Enter the message to be signed: ")

   key = generate_rsa_key_pair()

   private_key = key.export_key()

   public_key = key.publickey().export_key()

   signature = sign_message(message, key)


   print("Private Key:\n", private_key.decode())

   print("Public Key:\n", public_key.decode())

   print("Signature:\n", signature.hex())


elif operation.lower() == "verification":

   message = input("Enter the message: ")

   signature = bytes.fromhex(input("Enter the signature: "))

   public_key_str = input("Enter the RSA public key: ")

   public_key = RSA.import_key(public_key_str)
```

```
    if verify_signature(message, signature, public_key):

        print("Signature is valid.")

    else:

        print("Signature is invalid.")


else:

    print("Invalid operation. Please choose 'Generation' or
'Verification'.")



# MAC

from cryptography.hazmat.primitives import hmac

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.backends import default_backend


def generate_mac(message, algorithm):

    # Generate a secret key (for demonstration purposes only)

    secret_key = b'your_secret_key'


    # Initialize the HMAC object with the selected hash algorithm and
secret key

    mac_algorithm = hmac.HMAC(secret_key, algorithm,
backend=default_backend())


    # Feed the message into the MAC algorithm
```

```
    mac_algorithm.update(message.encode('utf-8'))

    # Generate the MAC value
    mac_value = mac_algorithm.finalize()

    return mac_value.hex()

# User interface
def main():
    # Prompt user for inputs
    message = input("Enter the message: ")
    print("Available MAC algorithms:")
    print("1. HMAC-SHA256")
    print("2. HMAC-SHA512")
    choice = int(input("Select MAC algorithm (1 or 2): "))

    if choice == 1:
        algorithm = hashes.SHA256()
    elif choice == 2:
        algorithm = hashes.SHA512()
    else:
        print("Invalid choice.")
```

```
        return

    # Generate MAC
    mac_value = generate_mac(message, algorithm)

    # Display MAC to the user
    print("Generated MAC:", mac_value)

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