program for Bob uses the RSA cryptosystem with a very large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends a message to Bob by representing each alphabetic character as an integer between 0 and 25 (A S 0, c, Z S 25) and then encrypting each number separately using RSA with large e and large n. Is this method secure? If not, describe the most efficient attack against this encryption method.

# Simple demonstration: why encrypting letters 0..25 separately with RSA is insecure.

# No external libs. Easy to run.

from math import gcd

# --- helper RSA functions (small-scale demo) ---

def encrypt\_block(m, e, n):

return pow(m, e, n)

def decrypt\_block\_with\_lookup(c, lookup):

# lookup: ciphertext -> list of possible plaintext values

return lookup.get(c, [])

# --- demo parameters (small n for easy reading) ---

# In practice n is huge; the attack still works because domain size is 26.

p, q = 61, 53

n = p \* q # 3233

e = 17 # public exponent (coprime with phi)

phi = (p-1)\*(q-1)

# (Optional) compute private exponent d just to demonstrate real decryption

def egcd(a,b):

if b==0: return (a,1,0)

g,x1,y1 = egcd(b, a%b)

return (g, y1, x1 - (a//b)\*y1)

g,x,y = egcd(e, phi)

d = x % phi

print("Demo RSA params: n =", n, "e =", e, " (private d =", d, ")")

print()

# --- plaintext -> numeric mapping ---

def text\_to\_nums(txt):

out = []

for ch in txt.upper():

if 'A' <= ch <= 'Z':

out.append(ord(ch) - ord('A'))

else:

out.append(None) # non-letter (we keep it None)

return out

def nums\_to\_text(nums):

s = []

for v in nums:

if v is None:

s.append('?')

else:

s.append(chr(v + ord('A')))

return ''.join(s)

# Example message

message = "HELLO" # plaintext we will "encrypt" letter-by-letter

nums = text\_to\_nums(message)

print("Plaintext:", message)

print("Numeric mapping:", nums)

# --- Encrypt each letter separately ---

cipher\_blocks = []

for m in nums:

if m is None:

cipher\_blocks.append(None)

else:

cipher\_blocks.append(encrypt\_block(m, e, n))

print("Cipher blocks:", cipher\_blocks)

# ----------------------

# ATTACK: precompute mapping for m = 0..25

# ----------------------

lookup = {} # ciphertext -> list of possible m values

for m in range(26):

c = encrypt\_block(m, e, n)

lookup.setdefault(c, []).append(m)

print("\nPrecomputed lookup table (cipher -> plaintext values):")

for c, ms in lookup.items():

print(f" {c:4} -> {ms}")

# Use lookup to recover plaintext from ciphertext blocks

recovered\_nums = []

for c in cipher\_blocks:

if c is None:

recovered\_nums.append(None)

else:

candidates = decrypt\_block\_with\_lookup(c, lookup)

if len(candidates) == 1:

recovered\_nums.append(candidates[0])

else:

# collision or ambiguity (rare); pick first candidate but show warning

recovered\_nums.append(candidates[0])

print("\nWarning: ciphertext", c, "maps to multiple plaintexts:", candidates)

recovered\_text = nums\_to\_text(recovered\_nums)

print("\nRecovered numeric mapping:", recovered\_nums)

print("Recovered plaintext (by lookup):", recovered\_text)

# --- For comparison: decrypt with private key d (true RSA decryption) ---

decrypted\_by\_d = []

for c in cipher\_blocks:

if c is None:

decrypted\_by\_d.append(None)

else:

decrypted\_by\_d.append(pow(c, d, n))

print("\nDecrypted via private key d (should match original numbers):", decrypted\_by\_d) 