Part 1: RSA Implementation

The RSA implementation follows the standard algorithm with a 256-bit modulus created from two 128-bit primes. Key generation ensures both primes p and q have their two leftmost bits set and are coprime to e=65537. For encryption, the code processes message.txt in 128-bit blocks, padding each block with zeros on the right if needed to make complete blocks, then prepending 128 zeros by left-shifting (block << 128) to create 256-bit blocks. Each padded block M is encrypted using C = M^e mod n. For decryption, the Chinese Remainder Theorem optimization is implemented by computing d_p = d mod (p-1) and d_q = d mod (q-1), then calculating m_p = C^d_p mod p and m_q = C^d_q mod q. These intermediate values are combined using the formula M = m_q + h*q where h = q_inv * (m_p - m_q) mod p. The 128 zero bits are removed by right-shifting (M >> 128), and any trailing padding zeros are stripped to recover the original message.

Part 2: Breaking RSA with Small Exponents

The RSA breaking implementation demonstrates why small values of e (like e=3) are insecure. The code generates three different RSA key pairs all using e=3, encrypts the same message with each key pair, and then exploits this vulnerability to recover the original message. When e=3, the Chinese Remainder Theorem can be applied to find M^3 mod N (where N is the product of the three moduli). Since M^3 < N when using sufficiently large moduli, M^3 mod N equals M^3, allowing direct cube root calculation to recover M. The implementation uses the solve_pRoot function to accurately calculate this cube root since Python's native functions lack sufficient precision. This attack works because a small exponent like e=3 doesn't sufficiently scramble the message when the same message is encrypted with multiple keys sharing that exponent.