Solutions to Daily Due Sep 21 (AI generated)

Math 4440/5440, Coding and Cryptography, Fall 2026

Generated by Claude for Katherine E. Stange
(conventions follow the Fall 2024 lecture notes of Sept 27 and 30;
all computations computer-verified)

Problem 1: malleability of El Gamal

What Bob decrypts

Recall the set-up. Bob's private key is b and his public key is $B = g^b \bmod p$. To send m, Alice picks an ephemeral k and sends

$$(K, c) = \left(g^k,\ mB^k\right) \pmod{p}.$$

Bob decrypts by computing $K^{-b}c$, which works because $K^b = g^{kb} = B^k$:

$$K^{-b}c = K^{-b}mB^k = m.$$

Now suppose Oscar replaces (K, c) by (K, sc), for some s he chooses. Bob follows the same recipe and gets

$$K^{-b}(sc) = s\left(K^{-b}c\right) = \boxed{s\,m} \pmod{p}.$$

So **Bob decrypts** sm: Oscar's factor passes straight through the decryption.

Why that is bad, even though Oscar learns nothing

Oscar does not learn m — he has no way to compute it — but he controls the *change* to m exactly. If the message is a dollar amount, he can double it ($s = 2$); if he later learns the plaintext by other means, he knew the offset all along. This is called **malleability**, and it is an *active* attack: Oscar has to interact with the protocol and modify data in transit, not merely listen.

The underlying reason is that El Gamal encryption is *multiplicatively homomorphic*: the ciphertext of m_1m_2 is the componentwise product of the ciphertexts. That is a feature in some settings (it is the basis of homomorphic voting schemes) and a bug in most.

How to fight it

The general principle: encryption protects *confidentiality*, not *integrity*. If you want integrity you must add something.

- **Add a MAC or a signature.** Alice appends a message authentication code or a digital signature computed on the ciphertext (encrypt-then-MAC). Bob refuses to decrypt anything whose tag does not verify. Oscar cannot produce the tag for (K, sc) without the key.

- **Add redundancy the attacker cannot preserve.** Instead of encrypting m, encrypt $m \| H(m)$ for a cryptographic hash H, or a padded, structured encoding of m. When Bob decrypts sm, the structure is destroyed and he rejects the message. (This is why real RSA and El Gamal are always used with padding schemes such as OAEP, rather than "textbook" style.)
- **Hash the shared secret.** Use $c = m \oplus H(B^k)$ or a symmetric cipher keyed by $H(B^k)$, rather than multiplying by B^k. Multiplying by s then produces garbage rather than a predictable multiple. This is what real hybrid encryption (DHIES/ECIES) does, and it kills malleability together with several other problems.

The strongest formal goal here is called *non-malleability* (or IND-CCA2 security); textbook El Gamal is only IND-CPA secure, which is precisely the gap this problem exposes.

Problem 2: the RSA ciphertext chain, worked end to end

There is nothing to "solve" here, so what follows is one complete worked pass through the exercise with real numbers, small enough to print but full-size in structure. All numbers were computed in Sage/Python and checked.

(2) Setting up a key pair

Pick two 8-digit primes with `next_prime`:

$$p = 31415971, \qquad q = 27182839.$$

Then

$$n = pq = 853975281721669, \qquad \varphi(n) = (p-1)(q-1) = 853975223122860.$$

Note that $\varphi(n)$ was computed from the *formula*, not with `euler_phi(n)`; at cryptographic size the latter would have to factor n and would never finish.

Choose the encryption exponent $e = 65537$. It is prime, and $\gcd(65537, \varphi(n)) = 1$, so it is invertible — this is exactly why a prime e is a good bet (concept check from the post: a prime e fails to be invertible only in the unlikely event that e itself divides $\varphi(n)$). The decryption exponent is

$$d = e^{-1} \bmod \varphi(n) = 822637041460433,$$

computed with `inverse_mod(e, phi)`. Check: $ed \equiv 1 \pmod{\varphi(n)}$; in fact $ed = 1 + 63132\,\varphi(n)$.

Public key: $(n, e) = (853975281721669,\ 65537)$. **Private:** d (and keep $p, q, \varphi(n)$ secret too — any one of them gives d away).

(3)–(4) The message as an integer

Take the message `tacos`. The *Text to Integer* tool computes $\sum_i \mathrm{ord}(c_i)\, 255^i$, i.e. it reads the letters as digits base 255:

$$m = 116 + 97 \cdot 255 + 99 \cdot 255^2 + 111 \cdot 255^3 + 115 \cdot 255^4 = 488095816826.$$

Notice $m < n$, which is essential — RSA can only encrypt one residue modulo n, and this is why the post insists on at most six letters: six letters reach at most $255^6 - 1 \approx 2.7 \times 10^{14} < n$, while a seventh letter would push the integer past $255^6 \cdot 255 \approx 7 \times 10^{16} > n$, so it would wrap around modulo n and be undecryptable.

(5) Encrypting to someone's public key

With their public key (n, e) as above,

$$c = m^e \bmod n = 488095816826^{65537} \bmod 853975281721669 = 47632482136830.$$

In Sage: `power_mod(m, e, n)`. Post c on `#ciphertexts`.

(6) Decrypting

The recipient computes

$$c^d \bmod n = 47632482136830^{822637041460433} \bmod 853975281721669 = 488095816826 = m,$$

and the Integer to Text tool turns that back into `tacos`.

Why it works. Since $ed = 1 + k\varphi(n)$,

$$c^d = (m^e)^d = m^{ed} = m^{1+k\varphi(n)} = m \cdot \left(m^{\varphi(n)}\right)^k \equiv m \cdot 1^k = m \pmod{n}$$

by Euler's theorem (and the conclusion holds even for the rare m sharing a factor with n, by the Chinese remainder theorem argument in the textbook).

Things that go wrong, and what they teach

- *The message is too long.* If $m \geq n$ you recover $m \bmod n$, which is not your message. Real RSA splits long messages into blocks, or — far better — encrypts a symmetric key and sends the message under AES.
- *e not invertible.* Then no d exists; pick another e. This happens exactly when $\gcd(e, \varphi(n)) > 1$.
- *Primes too close together.* If p and q are near each other, Fermat factoring cracks n immediately: it searches $n + y^2 = x^2$ for small y, which succeeds fast precisely when p and q are close.
- *Short messages.* A six-letter word is a tiny message space, which is what the "small message" birthday attack from class exploits. Real RSA pads messages with random bits for exactly this reason.