Solutions to Daily Due Sep 28 (AI generated)

Math 4440/5440, Coding and Cryptography, Fall 2026

Generated by Claude for Katherine E. Stange
(all computations computer-verified)

Problem 1 was to watch the pre-recorded lecture; the solutions below cover problems 2–10.

Problem 2: one equation, two inverses

Suppose $ax + by = 1$. Reduce modulo b: the term by vanishes, leaving

$$ax \equiv 1 \pmod{b},$$

so x is the inverse of a modulo b. Now reduce the same equation modulo a instead: the term ax vanishes, leaving

$$by \equiv 1 \pmod{a},$$

so y is the inverse of b modulo a. One run of the extended Euclidean algorithm therefore hands you both inverses: x (reduced mod b) and y (reduced mod a).

Problem 3: Alice's RSA signature

$n = 3233 = 61 \cdot 53$, so $\varphi(n) = 60 \cdot 52 = 3120$. The private exponent is $d = 17^{-1} \bmod 3120 = 2753$ (check: $17 \cdot 2753 = 46801 = 15 \cdot 3120 + 1$).

Signing $m = 65$: Alice computes

$$s = m^d = 65^{2753} \equiv 588 \pmod{3233}.$$

Verifying as Bob, using only the public key (n, e):

$$s^e = 588^{17} \equiv 65 \pmod{3233} = m. \qquad \checkmark$$

Problem 4: existential forgery

Pick $s = 100$ and compute the "message" it signs:

$$m = s^e = 100^{17} \equiv 1773 \pmod{3233}.$$

Now the pair $(m, s) = (1773, 100)$ passes Bob's check ($s^e \equiv m$) even though Alice never signed anything. This is an *existential forgery*: the attacker produced a valid signature, but on a message they had no control over. It doesn't worry Alice much because 1773 is gibberish; in practice messages have structure (English text, a specific format, or, better, a hash of a meaningful document), and a random m will essentially never be a meaningful message. Signing $h(m)$ for a cryptographic hash h makes this airtight: the forger would need to find a meaningful document whose hash is exactly 1773, i.e. a preimage.

Problem 5: multiplying signatures

Signatures are multiplicative: $s_1 s_2 = m_1^d m_2^d = (m_1 m_2)^d$. So

$$s = s_1 s_2 = 588 \cdot 3065 \equiv 1439 \pmod{3233}$$

is a valid signature on $m_1 m_2 = 65 \cdot 42 = 2730$. Check: $1439^{17} \equiv 2730 \pmod{3233}$. ✓

The exploit. Anything Alice has signed, she has unwittingly also signed the products of: from signatures on m_1 and m_2 an attacker can produce a valid signature on $m_1 m_2$, on m_1^2, on $m_1 m_2^{-1}$, and so on, none of which Alice ever saw. The classic version is a *blinding* attack: to get Alice's signature on a message m she would refuse, pick a random r and ask her to sign the innocent-looking gibberish $m' = mr^e \bmod n$. Her signature is $s' = (mr^e)^d = m^d r$, and dividing by r gives m^d, a valid signature on m.

Why hashing stops it. If Alice signs $h(m)$ rather than m, then multiplying signatures gives a signature on $h(m_1)h(m_2)$, and for a good hash function there is no efficiently findable message m_3 with $h(m_3) = h(m_1)h(m_2) \bmod n$ (that would be a preimage). The multiplicative structure of RSA no longer lines up with any structure on messages.

Problem 6: the Hash Tools page

A hash function takes a message of any length and produces a fixed-length digest (SHA-256 gives 256 bits, which is 64 hex characters). It is deterministic, fast, and should be preimage resistant (given a digest you can't find a message), collision resistant (you can't find two messages with the same digest), and should show the avalanche effect (a tiny change in the message scrambles the whole digest). The page lets you hash strings with several algorithms, show the digest in hex or bits, and compare two messages bit by bit.

Problem 7: SHA-256 and the avalanche effect

SHA-256 digests (hex):

message	SHA-256
`friend`	`cde48537ca2c28084ff560826d0e6388b7c57a51497a6cb56f397289e52ff41b`
`fiend`	`17f6591caea018f62fc7279a40a9d320918662a63afe0a34a5c9c05eb49e57a6`
`united`	`ca52e7d1c18961a8c5b3947a54e48fbf4ee70b66ea413cf7814f1c962631fd58`
`untied`	`90d7cdb59ca3a2e732cb97eab386e72f9905b2f5c021f2f00d52ddedaa9e0cd0`
`awesome`	`705db0603fd5431451dab1171b964b4bd575e2230f40f4c300d70df6e65f5f1c`
`yuck`	`2fd30d0b9f81b20edf804ddb8e9a4b74c31d18aa50f6475a7466351511ae709e`
`looooooooong`	`0be28d045275aaeb44534d5ef9b34cec2cab699f0d4ab5d59bda33f7dcaad50a`
`short`	`f9b0078b5df596d2ea19010c001bbd009e651de2c57e8fb7e355f31eb9d3f739`

(Each hex digit stands for 4 bits; the page's "bits" view just expands these.) Number of the 256 bits that differ between each pair:

pair	differing bits
friend / fiend	120
united / untied	123
awesome / yuck	134
looooooooooong / short	125

This is exactly what we'd expect from the avalanche effect: for a good hash function, changing the input in *any* way (removing one letter, swapping two letters, or replacing the whole message) should make each output bit flip independently with probability 1/2, so about $256/2 = 128$ bits change, give or take $\sqrt{256}/2 = 8$ or so. All four counts sit comfortably in that range. Notice there is no sense in which "similar" inputs give "similar" digests.

Problem 8: the parity checksum

$h(m) =$ (sum of the bits of m) mod2, i.e. $h(m) = 0$ when m has an even number of 1s.

Preimage of 0: any message with an even number of 1s, e.g. $m = 0$, or $m = 11$, or $m = 0000$.

Collision with $m = 1010$: $h(1010) = 0$ (two ones), so any other string with an even number of 1s collides: e.g. 0101, 1100, 0000, 1111, 11.

(Indeed, half of all messages collide with 1010: this hash is as far from collision resistant as possible.)

Problem 9: the XOR-the-bytes hash

Write m as 8-bit blocks $B_1, B_2, \ldots, B_k$ and let $h(m) = B_1 \oplus B_2 \oplus \cdots \oplus B_k$.

Preimage: to hit any target digest D, take the single block $m = D$ (one byte), or $m = B \,\|\, (B \oplus D)$ for any byte B. For instance, a preimage of 00000000 is any block repeated twice, like the ASCII string `aa`.

Collision: XOR is commutative, so reordering the blocks doesn't change the digest: $h(\texttt{ab}) = h(\texttt{ba}) = 01100001 \oplus 01100010 = 00000011$. Appending two copies of any byte also changes nothing: $h(\texttt{abba}) = h(\texttt{ab}) \oplus h(\texttt{ba}) = 00000000$, and in general $h(m \,\|\, X \,\|\, X) = h(m)$ for any byte X, so m and $m \,\|\, XX$ always collide.

Avalanche? No. Flipping one bit of the message flips exactly one bit of the digest (XOR acts bit by bit), so similar messages have similar digests. The fact that preimages and collisions are trivial is one symptom; the lack of avalanche is another.

Problem 10: the birthday bound

A k-bit digest has $N = 2^k$ possible values. By the birthday paradox, after hashing about $\sqrt{N} = 2^{k/2}$ random messages you expect a collision (more precisely about $1.18\sqrt{N}$ for a 50% chance).

hash	k	messages for a collision
MD5	128	$2^{64} \approx 1.8 \times 10^{19}$
SHA-256	256	$2^{128} \approx 3.4 \times 10^{38}$

So a k-bit hash offers only $k/2$ bits of collision resistance; this is why 128-bit digests are now considered too short for collision resistance (and why MD5 was retired, though its actual break came from structural weaknesses, far faster than 2^{64}).

Preimages: no, the birthday trick doesn't help. For a preimage you must match one *specific* digest, not any pair among your own guesses, so you expect to need about $N = 2^k$ tries (2^{128} for MD5, 2^{256} for SHA-256). The birthday saving comes from the $\binom{M}{2}$ pairs among M messages all being chances for a collision; a preimage search has only M chances.