Paxson Spring 2017 — CS 161 Computer Security — Homework 3

Due: Wednesday, March 8, at 11:59pm

Instructions. This homework is due **Wednesday, March 8, at 11:59pm**. No late homeworks will be accepted. You *must* submit this homework electronically via Gradescope (not by any other method). When submitting to Gradescope, *for each question* your answer should either be a separate file per question, or a single file with each question's answer on a separate page. This assignment must be done on your own.

Problem 1 ***True-or-False Questions*** **(4 points)**

Answer each question. You don't need to justify or explain your answer.

(a) TRUE or FALSE Diffie–Hellman protects against eavesdroppers but is vulnerable to man-in-the-middle attacks.

> **Solution:** True. A man-in-the-middle attacker, Mallory, can intercept the public key that Alice sends to Bob (g^A) and the public key that Bob sends to Alice (g^B). Then, Mallory can send her own public key (g^M) to Alice and Bob, which they will interpret as (g^B and g^A) and use to establish shared keys with Mallory.

(b) TRUE or FALSE: Suppose there is a transmission error in a block B of ciphertext using CBC mode. This error propagates to every block in decryption, which means that the block B and every block after B cannot be decrypted correctly.

> **Solution:** False. Only B and the block after B are decrypted incorrectly.

(c) TRUE or FALSE: The IV for CBC mode must be kept secret.

> **Solution:** False. It can be public. For instance, it is normally sent in the clear along with the ciphertext, so any eavesdropper can see the IV—this does not cause any security problems.

(d) TRUE or FALSE: Alice and Bob share a symmetric key k. Alice sends Bob a message encrypted with k stating, "I owe you \$100", using AES-CBC encryption. Assuming AES is secure, we can be confident that an active attacker cannot tamper with this message; its integrity is protected.

Solution: False. An attacker can still modify the ciphertext sent, and there is no way for Bob to tell if the message has been modified.

Problem 2 ***Photo Retrieval System*** **(20 points)**

CalPix, a new social photo-sharing site, lets users upload photos to its servers and share them with their friends. For privacy, the service has implemented a form of access control: upon uploading a photo, it lets a user decide which other users can view it.

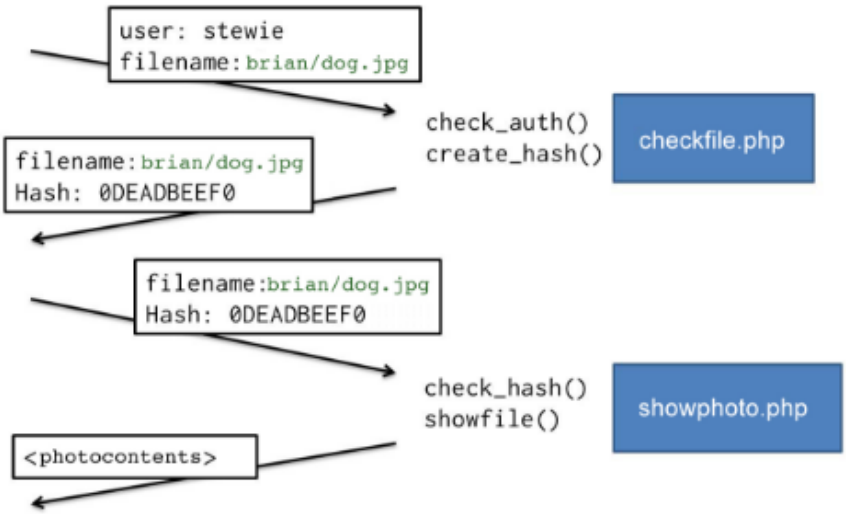
Photos are retrieved via calls to the `http://calpix.com/showphoto.php` page, which checks whether the given user should be able to view the photo, and if so, sends the photo file's contents.

The script `showphoto.php` accepts two URL parameters: `filename`, which is the filename of the photo to show, and `hash`, which is an authenticator. Photos are grouped into subdirectories by username, so the `filename` parameter is actually the path to the file, relative to the base photo directory (for example, `someuser/somephoto.jpg`).

Before showing any photo, `showphoto.php` checks the `hash` value sent. This hash value is the first five bytes (10 hex digits) of a SHA-256 digest of the concatenation of the filename and a secret encryption key (known only to the server).[1]

The page `http://calpix.com/checkfile.php` looks up the current logged-in user's identity and checks whether the user is allowed to access a particular photo file. If so, `checkfile.php` generates the correct hash value and redirects to `showphoto.php`. `checkfile.php` is able to generate the correct hash value since it runs on the server side and thus has access to the secret encryption key.

The diagram below sketches this protocol. User `stewie` has access rights for the file `brian/dog.jpg`.



[1] You may find this scheme peculiar, but in fact this example was inspired by a real life example, with the only significant change being that we're specifying the hash function as SHA-256, whereas in reality the function was MD5. We didn't want students distracted by possible weaknesses in MD5, which don't play a role here.

(a) Among the following primitives developed in class: (signatures, PRNGs, encryption, MAC, hashing, certificates), which primitive's usage and functionality does the above hashing scheme aim to achieve? Which of the three CIA goals discussed in lecture does the above hashing scheme aim to provide? State your answers to these two questions and explain your choices in one or two sentences.

Solution: The hashing scheme aims to achieve the same functionality as a *MAC*. It uses a secret key to generate a tag associated with a message (in this case, the filename) to prove that the message originated from someone possessing the secret key. (Here, the message is meant to be confirmed as coming from checkfile.php.) MAC is the best answer because we're in a *symmetric* key setting; other authenticity primitives, such as signatures, refer to an asymmetric key setting (hence those primitive's usage + functionality aren't good matches compared to MACs).

Verifying whether a message came from a particular entity corresponds to the goal of *authenticity*. Note that for the subpart of the question asking about CIA goals, we'll also accept *confidentiality* if students clearly justified their answer by stating that the system aims to keep users' photos private from unauthorized parties. However, in order to receive full credit for the entire problem, students would still need to cite MAC as the best fitting primitive; encryption is not a good fit because (1) no data is being encrypted and (2) the problem says nothing about a user sharing a secret encryption key with the calpix service.

(b) Why do we need a secret key as an input to the hash function?

Solution: Without the key, attackers can just compute the SHA-256 hash (also called the "digest") of the name of a file they want to see and include it in the URL. The secret ensures that this attack doesn't work, because the attackers don't possess all of the information necessary to compute the right digest.

Problem 3 ***Two Time Pads*** **(16 points)**

In class we learned about the One Time Pad (OTP) and Stream Ciphers, a practical form of using OTPs for encryption. As the name suggests, a one time pad should only ever be used *once* when encrypting messages. For this question, you'll explore what goes wrong if the same keystream (pad) is used multiple times to encrypt messages.

(a) Neo needs your help. He realizes that the evil empire of Caltopia is using a stream cipher to encrypt its communication. But luckily for us, the incompetent cryptographers they've employed have modified their stream cipher to use a single, fixed keystream when XOR'ing their messages to generate the resulting ciphertexts. Namely, the ciphertext for every message is generated as: $\forall i : C_i = K \oplus M_i$, where K is the same, fixed value for every message M_i.

Neo has stealthily recorded nine ciphertexts of messages that Caltopia has sent to one of their special agents, Mr. Redwood. Additionally, he has recovered a tenth ciphertext (the *target ciphertext*), which contains secret information that might help turn the tide against Caltopia. For your answer, submit the decrypted message of the *target ciphertext*.

All of the ciphertexts are hex-encoded strings and all of the original messages (including the target ciphertext's original message) contain only English letters, spaces, dashes ("-"), or exclamation marks ("!").

To help you out, Neo has written a string XOR function (shown below and written in Python 2.7). This function will return the character-by-character XOR of two *ASCII* strings. It also correctly accounts for different length strings; for two strings of different lengths, the longer string will be truncated to the shorter string's length before they are XOR'ed together. To use this function, you'll want to take the ciphertext strings and convert them to ASCII (e.g., in Python: `asciiStr = hexStr.decode('hex')`) before passing the strings into the XOR function. We will also post this code snippet on Piazza.

```
def strxor(a, b):
    """XOR two ASCII strings (trims the longer input if they're different lengths).

    If a and b are hexadecimal strings, convert them to ASCII before calling
    this function; e.g., strxor(a.decode('hex'), b.decode('hex'))
    """
    return "".join([chr(ord(x) ^ ord(y)) for (x, y) in zip(a, b)])  # Python 2.7
    # return bytes(x ^ y for (x, y) in zip(a, b))  # For Python 3
```

Hint: What happens when you XOR a letter with a space character? What happens if you XOR a letter with another letter?

Hint 2: After exploiting the many-time-pad usage in the ciphertexts below, there might be anywhere from three to six letters of the target ciphertext which you'll need to guess based on the rest of the message.

Ciphertext 1: 9dc26aa8bc3f39ba761ea6f583e705a9d1c7f2da07301c004812f2e9143
61f73361645611785f8b6a9d2af7083e0c70eb2abf14d10de3ce52bceac79a90dc24fe74b0
6838365

Ciphertext 2: 9dc26aa8b43d27b6701ea6a6c7ea09bb9fd0e494077818020d19e4e85c
3e137b6404177715d0fefea19bb46a8eebcc40e69db8402ccb34ad68cea47ae71bc102f60
45f958b658551

Ciphertext 3: 9ecf2fe6b43533f3761ef4edcaeb0abb8194e3dc126459124c12e5bd083
a10777e1959695ac4ffada581b37889f08217fa9db84f3a9f21a03adfed64b218cb1db3455
2da8f628e55584f

Ciphertext 4: 808a7ce9a67023bb6305f4f2cbea44bf9cc4fec616300a1e5814eff20b31
51557814456114ccefb1bfd2a67783a4cf09f59aec062bda77b620d3b963ae06c94ff74b51
94cc54b9640c6c173ce744f6

Ciphertext 5: 86c22fe6be7039bc765190c8e2af28b584daf0d1536411175913abea143
a037136045f6b5ad5e3aebc9ba97ec7e7ce15f0d2fb4725d332a168e4a278b301cb4ffa57

Ciphertext 6: 8cdc6afaa83238b77b51a7f2ccff44ae90d8fcdd1d7759184217aabd5c7f
51343650172e3385f8b6a59cac3993eccb13f19af94827da3bad29d5ed75a20dc00de1414
799847593

Ciphertext 7: 9ac363edbf3332f3221ebaa6d7e701fa86dde5d1533058560d40c8f5153
31566731e1a475ad6edb7a8d2b4708be1cc03f7d2b806699f77e568869e63a8188e4fb304
06da98719b4e454e1f

Ciphertext 8: e98a2fa8953f77aa6d04f4e3d5ea0afa9dddf1c053720b174540cff4187f
087b63505c6015d2acaaa497e74fb7a4cb13b286f04369d025ac2fcfa376ab48ec1dfc4354
9b817d9257

Ciphertext 9: 9bcf6ee4f13325aa7205bbe1d1ee14b294c6e494077f0d17410cf2bd183
0517a7904176514cafbfea49db03993eb8215e197b8760eef77b120c7b964af1dc30efd424
799987f855658551e2fa95afc

Target Ciphertext: 84d32feebe2223a16702a7a6c1e311bf81c6feda076359175f05ab
e8123b146636045f6b5adce3b9b980b33994e1c114fb9df60620d177872dd4a672ab0dd7
4fd14b5196

Submit the decrypted *Target Ciphertext* (as an ASCII string in English) as your answer to Part (a).

Solution:

1. Going off of the hint, we notice that when we XOR a letter with a space, the casing of the letter changes (i.e., it goes from lowercase to uppercase or vice-versa). On the other hand, if we XOR a letter with a letter, we get some random non-alphabet ASCII character.

2. For each of the nine ciphertexts, let's XOR it (call this ciphertext C*) with the other eight ciphertexts. If we look at the result of the eight XOR'ed ciphertexts and see which characters become an English letter in a *majority* of the eight XOR's, then we can be pretty confident that C* has a space character in those positions.
3. Once we know the space positions in each of the eight ciphertexts, we can then recover the majority of the many-time-keystream by XOR'ing the eight ciphertexts with their respective "space strings" (i.e., strings with space characters in the places we've uncovered and garbage in the other character positions).
4. Once we have the many-time-keystream, we just XOR this keystream with the target ciphertext: *My fortress blueprints are under the yogurt section in Berkeley Bowl*

(b) Submit the code you used to exploit the many-time-pad usage and decrypt the target ciphertext.

Solution: Come to office hours (no solution code released).

Problem 4 ***Finding Common Patients*** **(24 points)**

Caltopia has two hospitals: Bear Hospital and Tree Hospital, each of which has a database of patient medical records. These records contain highly sensitive patient information that should be kept confidential. For both hospitals, each medical record is a tuple (p_i, m_i), where p_i and m_i are strings that correspond to the patient's full name and medical record respectively; assume that every person in Caltopia has a unique full name. Thus, we can think of Bear Hospital's patient database as a list of tuples $(x_1, m_1), (x_2, m_2), ..., (x_n, m_n)$, where m_i is the medical information that Bear Hospital has for patient x_i. Similarly, we can think of Tree Hospital's database as a list $(y_1, m'_1), (y_2, m'_1), ..., (y_m, m'_m)$, where m'_i is a string that encodes the medical information that Tree Hospital has for the patient named y_i. Note that for a given patient, Tree Hospital and Bear Hospital might have different medical information.

The two hospitals want to collaborate on a way to identify which Caltopia citizens are patients at both hospitals. However, due to privacy laws, the two hospitals cannot share any plaintext information about patients (including their names) unless both hospitals know *a priori* that a patient has used both hospitals.

Thus, the two hospitals decide to build a system that will allow them to identify common patients of both hospitals. They enlist the help of Lady Olenna, who provides them with a trusted, third-party server S, which they will use to discover the names of patients who use both hospitals. Specifically, Bear Hospital will take some information from its patient database and transform it into a list $(x_1^*), (x_2^*), ..., (x_n^*)$ (where (x_i^*) is somehow derived from x_i (the patient's full name) and upload it to S. Similarly, Tree Hospital will take information from its patient database, transform it into a list $(y_1^*), (y_2^*), ..., (y_m^*)$, and upload this transformed list to S. Finally, S will compute a set of tuples $P = (i, j) : x_i = y_j$ of all pairs (i, j) such that $x_i^* = y_j^*$ and send P to both Bear Hospital and Tree Hospital. The two hospitals can then take their respective indices from the tuples in P to identify patients who use both hospitals.

We want to ensure three requirements with the above scheme: (1) if $x_i = y_j$, then $(i, j) \in P$, (2) if $x_i \neq y_j$, then it is very unlikely that $(i, j) \in P$, (3) even if Eve (an attacker) compromises S, she cannot learn the name of any patient at either hospital or the medical information for any patient. For this question, assume that Eve is a passive attacker who cannot conduct Chosen Plaintext Attacks; however, she does know the names of everyone in Caltopia, and there are citizens whose full names are a unique length.

Fill in your solutions below. Your solution can use the cryptographic hash SHA-256 and/or AES with one of the three block cipher encryption modes discussed in class; keep in mind that Eve can also compute SHA-256 hashes and use AES with any block cipher mode. You can assume that Bear Hospital and Tree Hospital share a key k that is not known to anyone else. You *cannot* use public-key cryptography or modular arithmetic.

(a) In the collaboration scheme described above, how should Bear Hospital compute x_i^* (as a function of x_i)? How should Tree Hospital compute y_i^* (as a function of y_i)? Specifically, your solution should define a function F that Bear Hospital will use to

transform x_i into x_i^*, and if relevant, a function G that Tree Hospital will use to transform y_i into y_i^*.

Solution: Correct solutions should extract only the patient names from each hospital's database and apply one of the following approaches/transformations to the names.

All valid approaches will need to (1) **deterministically** encrypt the patient names and (2) use a secure hash function to hide the length of the patient names; their approach can either hash first and then encrypt, or encrypt and then hash. Encryption needs to be done deterministically because the hospitals have no way to share the same IV/randomness for the same patient name (and all patient names are unique, so deterministic encryption isn't a problem).

Let $E_k(x)$ be a secure, deterministic encryption scheme and $H(x)$ be the SHA-256 hash function, then their function $F_k(x)$ (which will be the same for Bear and Tree Hospital) can either be:

$$F_k(x) = E_K(H(x))$$

or

$$F_k(x) = H(E_k(x)) or F_k(x) = H(x||k) or F_k(x) = H(k||x)$$

Then Bear Hospital could compute $x_i^* = F_k(x_i)$ and Tree Hospital could compute $y_i^* = F_k(y_i)$, where x_i and y_i are just the names of the patient.

Valid Choices for $E_k(x)$:

1. $E_k(x) = \text{AES-ECB}_k(x)$
2. Use AES-CBC with a fixed IV (say, all-zeros) for every message (patient name). $E_k(x) = \text{AES-CBC}_k(x)$, where IV is a fixed constant C.
3. Use AES-CTR with a fixed IV (say, all-zeros) for every message (patient name). $E_k(x) = \text{AES-CTR}_k(x)$, where IV is a fixed constant C.

Explanation for why you only use the patient name: In general, we should only give S the minimal amount of information it needs to do the computation. Conceptually, S only needs to compute a join on the patient names between Bear and Tree Hospital, so the $F_k(x)$ transformations should only use the patient names. Additionally, from a correctness standpoint, Bear Hospital and Tree Hospital can have different medical information for the same patient, so they cannot include the patient's medical information in the transformation.

Explanation for why hashing is needed: Need to hide the lengths of each patient's full name b/c security requirement 3 says that patients can have names of unique length and Eve knows the name of everyone. TODO: make it explanation more eloquent.

Explanation for why hashing alone isn't sufficient: Describe a dictionary attack on hashing-alone schemes. TODO: elaborate more.

(b) Explain why requirement (1) is met by your solution, i.e., explain why it is guaranteed that if $x_i = y_j$, then $x_i^* = y_j^*$ will hold. Explain your answer in one or two sentences.

Solution: In all of the schemes, Bear Hospital and Tree Hospital did the same things. Since all of the building blocks are deterministic, the outputs are completely determined by the inputs, and in particular if $x_i = y_j$ then $x_i^* = y_j^*$.

(c) Explain why requirement (2) is met by your solution, i.e., if $x_i \neq y_j$, explain why it is unlikely that $x_i^* = y_j^*$. Explain your answer in one or two sentences.

Solution: This amounts to showing that collisions are unlikely in F. The analysis will depend upon the particular scheme.

SHA-256 is collision-resistant, so collisions in it are unlikely: finding a collision would show that SHA-256 is broken, so since we believe that SHA-256 is secure, it's unlikely that there will be any collisions among the Bear Hospital and Tree Hospital patients. Also, AES-ECB and the AES block cipher are invertible (one-to-one) functions (this must be true, because if we had $E_k(x) = E_k(y)$ for $x \neq y$, we wouldn't be able to correctly decrypt), so they cannot introduce collisions either.

(d) Explain why requirement (3) is met by your solution, i.e., if S is compromised by Eve, then the information known to S does not let Eve learn any patient information (neither the names of patients at a particular hospital nor the medical history for any patient). Explain your answer in one or two sentences.

Solution: Most of the schemes contain a layer of encryption around the message M or its hash $H(M)$. Intuitively, this encryption provides confidentiality. Because there is no IV, if we encrypt the same message twice, we'll get the same ciphertext, but that is desirable in this application. Also, because there is no IV, if we encrypt two messages which have the same first block, the first block of the ciphertext will also be identical, but this turns out to cause no harm: if we are encrypting a hash, it is very unlikely that two different hashes will happen to start with the same 16 bytes (this has probability $1/2^{128}$); and if we are encrypting the patient information and then hashing the ciphertext, then the hash conceals whether or not two ciphertexts start with the same first block.

Problem 5 ***Why do RSA signatures need a hash?*** **(20 points)**

This question explores the design of standard RSA signatures in more depth. To generate RSA signatures, Alice first creates a standard RSA key pair: (n, e) is the RSA public key and d is the RSA private key, where n is the RSA modulus. For standard RSA signatures, we typically set e to a small prime value such as 3; for this problem, let $e = 3$.

To generate a standard RSA signature S on a message M, Alice computes $S = H(M)^d \bmod n$. If Bob wants to verify whether S is a valid signature on message M, he simply checks whether $S^3 = H(M) \bmod n$ holds. Analogous to RSA encryption, d is a private key known only to Alice and $(n, 3)$ is a publicly known verification key that anyone can use to check if a message was signed using Alice's private signing key.

For this question we'll now explore why RSA signatures use a hash function to compute the signatures. Suppose RSA signatures skipped using a hash function and just used M directly, so the signature S on a message M is $S = M^d \bmod n$. In other words, if Alice wants to send a signed message to Bob, she will send (M, S) to Bob, where $S = M^d \bmod n$ is computed using her private signing key d.

(a) With this simplified RSA scheme, how can Bob verify whether S is a valid signature on message M? In other words, what equation should he check, to confirm whether M, S was validly signed by Alice? You don't need to justify your answer; just show the equation.

> **Solution:** $S^3 = M \bmod n$.

(b) Mallory learns that Alice and Bob are using the simplified (hash-less) signature scheme described above and decides to trick Bob. Mallory wants to send some (M, S) to Bob that Bob will think is from Alice, even though Mallory doesn't know the private key. Explain how Mallory can find M, S such that S will be a valid signature on M.

You should assume that Mallory knows Alice's public key n, but not Alice's private key d. She can choose both M and S freely. The message M does not have to be chosen in advance and can be gibberish.

Hint: If Mallory chooses M and then tries to find a corresponding S, she'll be at a dead-end, because finding S requires inverting a one-way function (cubing modulo n), and we know that is hard without knowledge of the trapdoor (the private key d). So instead, she should ...

> **Solution:** Mallory should choose some random value to be S and then compute $S^3 \bmod n$ to find the corresponding M value. This M, S pair will satisfy the equation in part (a).
>
> **Alternative solution:** Choose $M = 1$ and $S = 1$. This will satisfy the

equation.

(c) Sameer is holding an auction. Alice and Bob will submit signed bids to the auctioneer Sameer, signed using this simplified RSA signature scheme. The message M is an integer that is their bid (in dollars), and they will send just their signature on M, signed using this simplified RSA scheme. Sameer will accept whichever bid is highest and expect that person to pay up however much they bid.

Mallory wants to mess with Bob (her rival). So, when Bob forms his bid M and sends Sameer the signed bid $S = M^d \bmod n$, Mallory intercepts the message from Bob containing S. Mallory would like to tamper with S to form a new signature S' that corresponds to a bid for $64\times$ as much as Bob's original bid, to force Bob to win the auction and pay through the nose for it. More precisely, she'd like to find a value S' such that S' is a valid signature on $64M$, so she can replace S with S' and forward the result onto Sameer. Help Mallory out: show how she can compute such an S'.

(Assume that M is small enough that $64M < n$, so $64M$ does not wrap around modulo n.)

Solution: Multiply S by 4 to give you $S' = 4S \bmod n$. To verify the signature, Sameer will check that $64M = (S')^3 \bmod n$. Since $S = M^d \bmod n$, we have $S' = 4M^d \bmod n$, so

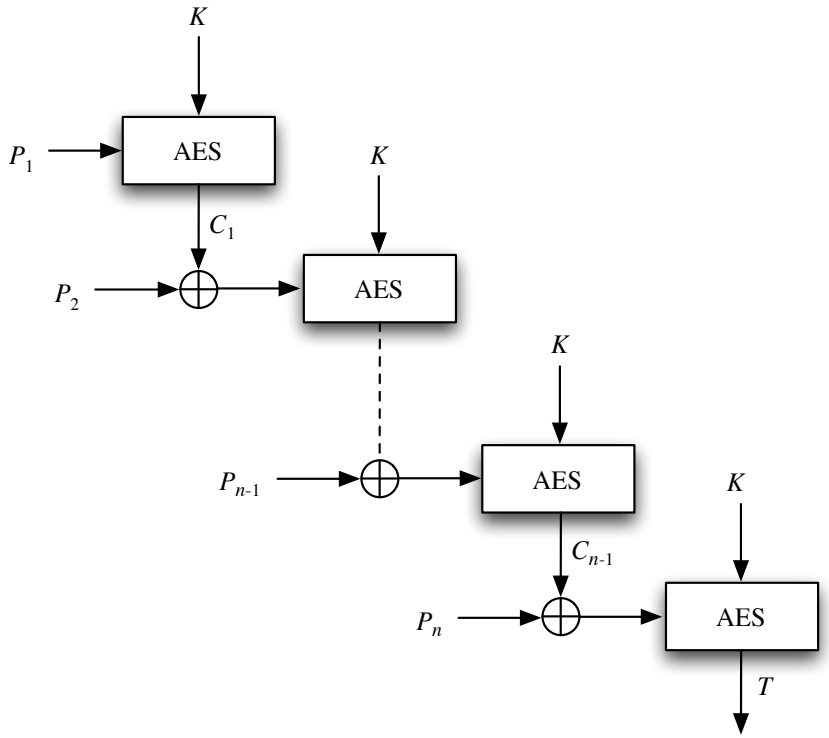
$$(S')^3 = (4 \times M^d)^3 = 4^3 \times (M^d)^3 = 64M \bmod n.$$

(d) Are your attacks in parts (b) and (c) possible against the real RSA signature scheme (the one that includes the cryptographic hash function)? Why or why not?

Solution: These attacks are not possible. A hash function is one way, so the attack in part (b) won't work: we can pick a random S and cube it, but then we'd need to find some message M such that $H(M)$ is equal to this value, and that's not possible since H is one-way. The attack in part (c) won't work, since given $H(M)$, we'd need to find a message M' such that $H(M') = 64 \times H(M)$, which is also infeasible (since H is one-way).

Comment: This is why the real RSA signature scheme includes a hash function: exactly to prevent the attacks you've seen in this question.

Problem 6 ***MAC Attack*** **(20 points)**



Consider the MAC algorithm shown in the diagram above. Each P_i is the ith block of a given message. At each stage, we encrypt the XOR of the previous stage and the next message block using the key K.

This algorithm is quite similar to AES-EMAC (shown in lecture and in the lecture notes), but differs in final stage, by using the same key as in the earlier stages, not including a second invocation of AES.

Suppose Mallory observes two single-block messages, M_1 and M_2, and the corresponding tags for these, T_1 and T_2. Show that Mallory can construct a message M_3 for which Mallory knows the associated tag T_3, even though Mallory does not know K.

Solution: We have:

$$\begin{aligned} T_1 &= \text{AES}_K(M_1) \\ T_2 &= \text{AES}_K(M_2) \end{aligned}$$

Consider a two-block message $M_3 = M_1||X$. That is, M_3 is M_1 plus a suffix X. Its tag will be:

$$\begin{aligned} T_3 &= \text{AES}_K(\text{AES}_K(M_1) \oplus X) \\ &= \text{AES}_K(T_1 \oplus X) \end{aligned}$$

Now if Mallory uses $X = T_1 \oplus M_2$, then we have:

$$\begin{aligned} T_3 &= \text{AES}_K(T_1 \oplus T_1 \oplus M_2) \\ &= \text{AES}_K(M_2) \\ &= T_2 \end{aligned}$$

Thus, even though Mallory doesn't know K, she knows that the tag for M_3 will be T_2.

Some students made the nice observation that in fact Mallory can proceed even without observing M_2 and its tag T_2. The same argument as above works if $M_3' = M_1||(T_1 \oplus M_1)$, where now the tag will be:

$$\begin{aligned} T_3' &= \text{AES}_K(T_1 \oplus T_1 \oplus M_1) \\ &= \text{AES}_K(M_1) \\ &= T_1. \end{aligned}$$