CS 161 Fall 2024 — Introduction to Computer Security — Midterm

Name: _____

Student ID: _____

Question:	1	2	3	4
Points:	0	8	20	19

Question:	5	6	7	Total
Points:	16	19	18	100

This exam is 110 minutes long. For questions with **circular bubbles**, select only one choice.

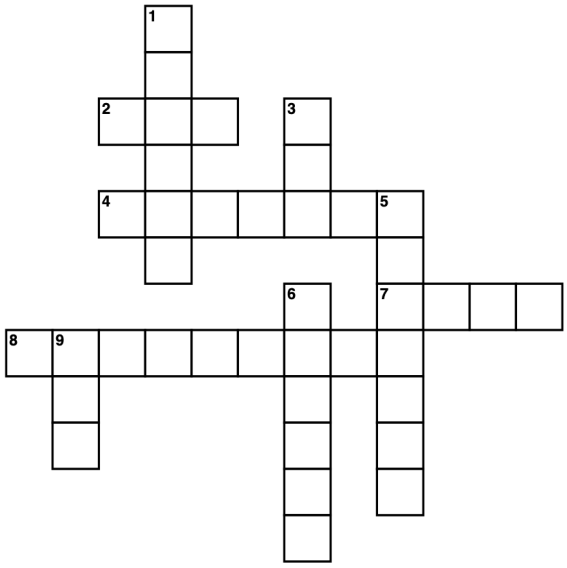
- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)
- ◍ Don't do this (it will be graded as incorrect)

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares (completely filled)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we may grade the worst interpretation.

Pre-exam activity (0 points):



Across:
(2) Block cipher previously known as Rijndael
(4) EvanBot's favorite food (singular)
(7) Randomizes addresses for each program execution
(8) $x \neq y$, $H(x) = H(y)$

Down:
(1) Hard to find x given $H(x)$
(3) Created by Rivest, Shamir, and Adleman
(5) The CS 161 mascot
(6) Used to encrypt and decrypt messages
(9) Perfectly-secure encryption

Q1 *Honor Code* **(0 points)**

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam.

Read the honor code above and sign your name: _____

Q2 *True/False* **(8 points)**

Each true/false is worth 1 point.

Q2.1 The employee entrance to the Caltopian Post Office is protected by a 20-digit keypad code that changes daily.

True or False: This violates Consider Human Factors.

○ True ○ False

Q2.2 Caltopian Army counterintelligence installs devices on all secure networks to look for abnormally high levels of outgoing data.

True or False: This is an example of Detect If You Can't Prevent.

○ True ○ False

Suppose we have a little-endian C program with a local variable `char easter_egg[32]`. Consider the following possible GDB output after running the command `x/8wx easter_egg`:

```
0xffffff000: 0x486F6D65 0x20706167 0x6520736F 0x75726365
0xffffff010: 0x20436865 0x636B2047 0x69744875 0x20707232
```

Q2.3 True or False: The last four bytes of `easter_egg` equal the word `0x20707232`.

○ True ○ False

Q2.4 True or False: The stack is always marked non-executable when non-executable pages is enabled.

○ True ○ False

Q2.5 True or False: When the ESP is moved up during function return, all memory that ends up below the ESP is set to zero.

○ True ○ False

Q2.6 True or False: One-time pads require both a random key and a random IV to be secure.

○ True ○ False

Q2.7 True or False: Small changes to the input of a hash function usually result in only minor changes to its output.

○ True ○ False

Q2.8 True or False: The generator and modulus (g, p) in Diffie-Hellman must not be reused between key exchanges.

○ True ○ False

Q3 *EvanBot's Folly: Memory Safety* **(20 points)**

EvanBot accidentally deleted the C library files responsible for `fgets`! They've taken it upon themselves to code a replacement:

```
1  void efgets(char* ptr, size_t maxlen) {
2      int c, i;
3
4      for (i = 0; i < maxlen; i++) {
5          c = fgetc(stdin); // Get one character from stdin
6          if (c == EOF)
7              break;
8
9          ptr[i] = c;
10     }
11     ptr[maxlen] = '\0';
12 }
```

Assumptions:

- All memory safety defenses are disabled.
- There is no compiler padding.
- You may use `SHELLCODE` as a 16-byte shellcode.
- `efgets` can be called like a C standard library function (similar to `gets`, etc).

EvanBot creates a sample program to demonstrate the use of `efgets`:

```
1  int main() {
2      char buf[16];
3      efgets(buf, 24);
4      return 0;
5  }
```

Q3.1 (2 points) Assume you have run this sample program using GDB and paused on **Line 3** of `main` (before `efgets` is called). You run the `info frame` command and see the following output (some lines excluded for brevity):

```
eip = 0x0100adb0 in main (sample.c:3); saved eip = 0x0100adfc
...
Saved registers:
ebp at 0xffffb550, eip at 0xffffb554
```

What is the address where **RIP of `main`** is stored on the stack?

- ○ `0x0100adb0`
- ○ `0xffffb550`
- ○ `0xffffb558`
- ○ `0x0100adfc`
- ○ `0xffffb554`
- ○ `0xffffb55c`

The `efgets` function is reproduced for your convenience:

```
1  void efgets(char* ptr, size_t maxlen) {
2      int c, i;
3
4      for (i = 0; i < maxlen; i++) {
5          c = fgetc(stdin); // Get one character from stdin
6          if (c == EOF)
7              break;
8
9          ptr[i] = c;
10     }
11     ptr[maxlen] = '\0';
12 }
```

Q3.2 (3 points) You now run this program on a second machine where the RIP of `main` is stored at a different address, specifically, at address `0xffffff18`. (Ignore the GDB output in Q3.1 from here on. Don't forget that `SHELLCODE` is 16 bytes long.)

Select the correct option for an input to `stdin` that causes the program to execute `SHELLCODE`.

- ○ `SHELLCODE + 'A'*4 + '\x18\xff\xff\xff'`
- ○ `SHELLCODE + '\x18\xff\xff\xff'`
- ○ `'A'*20 + SHELLCODE`
- ○ `SHELLCODE + 'A'*4 + '\x04\xff\xff\xff'`

Q3.3 (2 points) Assuming that the correct exploit was given in the previous subpart, when will `SHELLCODE` be executed?

- ○ Immediately after `fgetc` returns
- ○ During the execution of `efgets`
- ○ Immediately after `efgets` returns
- ○ Immediately after `main` returns

Q3.4 (2 points) Which of the following changes would prevent the correct exploit from Q3.2 (without modifications) from executing `SHELLCODE`? Select all that apply.

- ☐ Enabling stack canaries
- ☐ Removing the `EOF` check on Line 7
- ☐ Enabling non-executable pages
- ☐ None of the above

Impressed with your previous success, EvanBot decides to issue you a new challenge program **with ASLR enabled**!

```
1 void run_test() {
2     char buf[16];
3     efgets(buf, 16);
4 }
5
6 int main() {
7     run_test();
8     return 0;
9 }
```

Assume that EvanBot's program is the only program that modifies stack memory in this system, including memory that ends up below the ESP.

Stack at Line 3 of `efgets`

RIP of `main`
(1)
RIP of `run_test`
SFP of `run_test`
(2)
`maxlen`
(3)
RIP of `efgets`
SFP of `efgets`
`c`
`i`

Q3.5 (2 points) Fill in the stack diagram, assuming the program is paused on the third line of `efgets`.

- ○ (1) SFP of main (2) `buf` (3) `ptr`
- ○ (1) SFP of main (2) `buf` (3) `c`
- ○ (1) `buf` (2) `c` (3) `ptr`
- ○ (1) `buf` (2) `ptr` (3) `c`

Q3.6 (2 points) Which vulnerability is present in the code?

- ○ Off-by-one
- ○ ret2ret
- ○ Signed/unsigned vulnerability
- ○ Return-oriented programming

Q3.7 (7 points) [Warning: This part is hard; consider coming back to it if you are stuck.]

Give an input to `stdin` that would cause `SHELLCODE` to be executed by the new program with probability $\geq 1/256$.

The attack will be successful (i.e., `SHELLCODE` will be executed) if the least significant byte of the **value initially stored in** SFP of `run_test` is equal to which of the following?

- ○ `0x00`
- ○ `0x12`
- ○ `0x24`
- ○ `0x36`
- ○ `0x04`
- ○ `0x20`
- ○ `0x32`
- ○ `0x64`

Q4 *Copycat - Memory Safety* (19 points)

```
1  void foo(uint8_t offset, char* buf_ptr) {
2      int x = 0x11223344;
3
4      memcpy(buf_ptr + offset, &x, 8);
5  }
6
7  int main() {
8      char buf[132];
9      int8_t offset = 0;
10
11     gets(buf);
12     fread(offset, 1, 1, stdin);
13
14     if (offset > 20)
15         return 0;
16
17     foo(offset, buf);
18
19     return 0;
20 }
```

Stack at Line 4

RIP of `main`
SFP of `main`
Canary
`buf`
`offset`
(1)
(2)
RIP of `foo`
SFP of `foo`
(3)
`x`

Assumptions:

- Stack canaries are enabled, but all other memory safety defenses are disabled.
- There is no compiler padding.
- There is shellcode already stored at address `0xDEADBEEF`.
- `uint8_t` and `int8_t` are the C types for an 8-bit unsigned/signed integer, respectively.

Q4.1 (2 points) What values go in blanks (1) through (3) in the stack diagram above?

- ○ (1) Canary (2) `offset` (3) `buf_ptr`
- ○ (1) Canary (2) `buf_ptr` (3) `buf_ptr`
- ○ (1) `offset` (2) `buf_ptr` (3) Canary
- ○ (1) `buf_ptr` (2) `offset` (3) Canary

Q4.2 (2 points) Which vulnerability is present in the code?

- ○ Off-by-one
- ○ Signed/unsigned vulnerability
- ○ ret2ret
- ○ Out-of-bounds read

Q4.3 (1 point) True or False: The value of the stack canary is the same for each different function frame in the same program execution.

○ True ○ False

In the next two subparts, provide inputs that would cause the program to execute `SHELLCODE`.

If a part of the input can be any non-zero value, use `'A'*n` to represent `n` bytes of garbage.

Q4.4 (3 points) Input to `gets` at Line 11:

Q4.5 (5 points) Input to `fread` at Line 12 (**in hexadecimal**):

Q4.6 (3 points) Which of the following modifications would prevent this exploit (without any modifications) from working? Select all that apply.

☐ Generating the canary such that its least-significant byte is a null terminator.

☐ Changing the type of `offset` on Line 9 to `uint8_t`.

☐ Changing the condition on Line 14 to `offset > 128`.

☐ None of the above

Q4.7 (3 points) Select all values for the size of `buf` such that it is still possible to exploit this code, assuming you are able to pick new inputs.

☐ 4 ☐ 24 ☐ 128

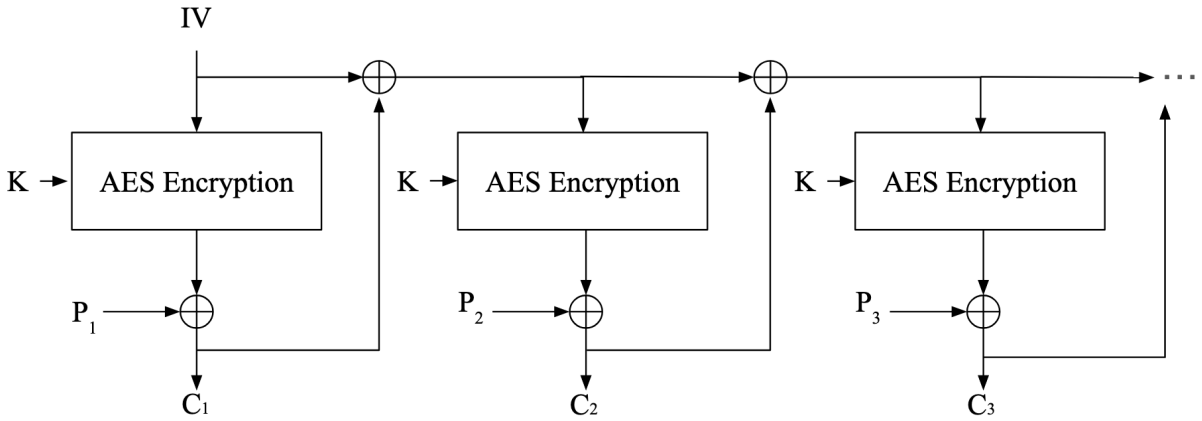
☐ 20 ☐ 64 ☐ 256

Q5 *AES-RFM — Symmetric Cryptography* **(16 points)**

EvanBot invents a new block cipher mode of operation: AES Repeated Feedback Mode.

The encryption formulas for AES-RFM are as follows:

$$C_1 = E_K(IV) \oplus P_1$$
$$C_i = P_i \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$$



Q5.1 (2 points) Select the correct decryption formula for the i-th ($i \geq 2$) plaintext block in AES-RFM.

- ○ $P_i = C_i \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$
- ○ $P_i = D_K(C_i) \oplus IV \oplus C_1 \oplus \cdots \oplus C_{i-1}$
- ○ $P_i = C_{i-1} \oplus E_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$
- ○ $P_i = C_i \oplus D_K(IV \oplus C_1 \oplus \cdots \oplus C_{i-1})$

Q5.2 (3 points) Alice has a 4-block message (P_1, P_2, P_3, P_4). She encrypts this message with AES-RFM and obtains the ciphertext $C = (IV, C_1, C_2, C_3, C_4)$, which she then sends to Bob.

During transit, network errors flip a single bit in C_1. That is, Bob receives the ciphertext $C' = (IV, C_1 \oplus 1, C_2, C_3, C_4)$.

What message will Bob compute when he decrypts the modified ciphertext C'?

G represents some unpredictable "garbage" output (individual G blocks do not necessarily have the same value).

- ○ (G, G, G, G)
- ○ $(P_1 \oplus 1, G, G, G)$
- ○ (G, P_2, P_3, P_4)
- ○ $(P_1 \oplus 1, P_2, G, G)$
- ○ (P_1, G, G, G)
- ○ $(P_1, P_2 \oplus 1, G, G)$

Alice has a 3-block message (P_1, P_2, P_3). She encrypts this message with AES-RFM and obtains the ciphertext $C = (IV, C_1, C_2, C_3)$.

As Mallory, you will modify the ciphertext C in transit to C', and you wish to choose C' so it decrypts to $P' = (X, Y, 1)$ (**where X and Y can be any value, garbage or otherwise**). In other words, you want to ensure that the last block of P' will be 1, but you don't care what the first two blocks of P' turn out to be. You have access to the original message (P_1, P_2, P_3).

Q5.3 (6 points) Select values for the modified ciphertext $C' = (IV', C'_1, C'_2, C'_3)$ such that Bob will decrypt C' to $P' = (X, Y, 1)$.

Each value below will be represented as the XOR of multiple variables. Select as many as you need. For example, if you want to set $IV' = P_1 \oplus C_2$, then bubble in P_1 and C_2.

IV' is equal to the XOR of:

☐ P_1 ☐ P_2 ☐ P_3 ☐ IV ☐ C_1 ☐ C_2 ☐ C_3 ☐ 1

C'_1 is equal to the XOR of:

☐ P_1 ☐ P_2 ☐ P_3 ☐ IV ☐ C_1 ☐ C_2 ☐ C_3 ☐ 1

C'_2 is equal to the XOR of:

☐ P_1 ☐ P_2 ☐ P_3 ☐ IV ☐ C_1 ☐ C_2 ☐ C_3 ☐ 1

C'_3 is equal to XOR of:

☐ P_1 ☐ P_2 ☐ P_3 ☐ IV ☐ C_1 ☐ C_2 ☐ C_3 ☐ 1

Q5.4 (5 points) Which values of P' can Mallory cause Bob to decrypt to, given that she can modify C and knows the original value of P? As in the previous subpart, X and Y represent values that Mallory doesn't need to control or predict and might be garbage.

Assume that none of the original P_i values were equal to 1.

- ☐ $(X, 1, Y)$
- ☐ $(P_1, 1, P_2)$
- ☐ $(P_1, 1, X)$
- ☐ $(P_1, P_2, 1)$
- ☐ $(1, P_2, P_2)$
- ☐ $(1, 1, X)$

Q6 *To HMAC and Back — Cryptography* **(19 points)**

For each of the following subparts, indicate whether the given construction is an EU-CPA secure MAC.

Assume that the message M is variable length and does not require padding.

Q6.1 (2 points) $\mathsf{MAC}(K, M) = H(M) \oplus H(K)$.

○ Secure ○ Insecure

Q6.2 (2 points) $\mathsf{MAC}(K_1, K_2, M) = H(K_2 \| H(K_1 \| M))$.

○ Secure ○ Insecure

Q6.3 (2 points) $\mathsf{MAC}(K, M) = \mathsf{HMAC}(K, M) \| M$.

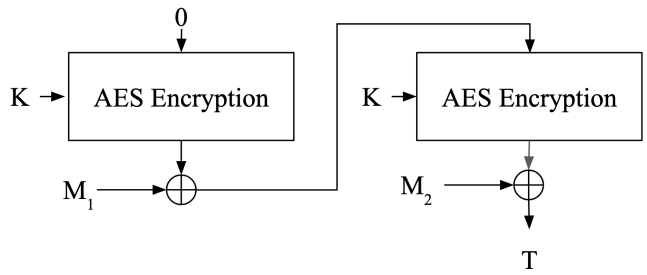
○ Secure ○ Insecure

Q6.4 (2 points) $\mathsf{MAC}(K, M) = (IV, C_n)$ where C_n is the last block of the AES-CBC encryption of $H(M)$ under key K, and IV is the corresponding randomly-generated IV.

For this subpart only, assume the cryptographic hash function H has an output size of 128 bits.

○ Secure ○ Insecure

Q6.5 (5 points) $\mathsf{MAC}(K, M) = C_n$, where C_n is the last block of the encryption of M with AES-CFB under key K and $IV = 0$. For example, $\mathsf{MAC}(K, [M_1, M_2]) = M_2 \oplus E_K(M_1 \oplus E_K(0))$:



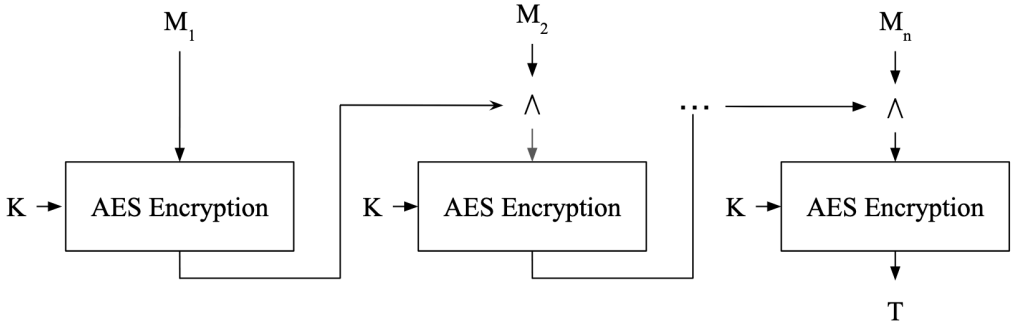
Given $M = (M_1, M_2)$ and its MAC T, provide a new **two-block** message $M' = (M'_1, M'_2)$ and its MAC T'. (You must not use $M' = M$.)

Provide a value for M':

Provide a value for T':

Q6.6 (6 points) Define AES-AND-MAC$(K, M) = C_n$, where $C_i = E_K(M_i \wedge C_{i-1})$ and $C_1 = E_K(M_1)$.

$\wedge$ represents **logical AND** between two 128-bit blocks (bitstrings).



You will describe an attack on this MAC. First, you request a MAC over a **one-block** message M of your choosing.

Provide a value for M:

You then receive a MAC T over M. Given (M, T) from the previous step, provide a new **two-block** message M' with MAC T' that you can compute from the information available to you (without knowing the key K).

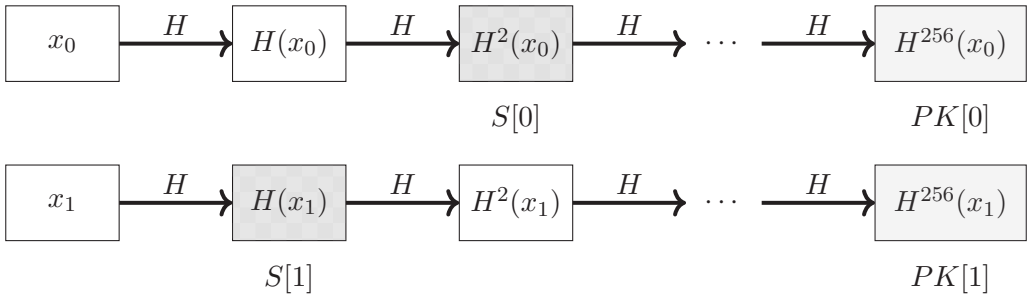
HINT: M must not equal M', but T can equal T'.

Provide a value for M':

Provide a value for T':

Q7 *Opaque Only Once — Digital Signatures* **(18 points)**

The rise of quantum computing worries EvanBot, who decides to invent a post-quantum signature scheme using only hash functions.



Pictured: A signature over the two-byte message $M = [$`0x02`, `0x01`$]$, with signature and public key: $S = [H^2(x_0), H(x_1)]$, $PK = [H^{256}(x_0), H^{256}(x_1)]$.

Clarification after exam: The example pictured above has a bug: it doesn't hash the message. The real scheme hashes the message first and then signs the bytes of the hash.

Key Generation:

1. Generate a list of 32 randomly-generated 256-bit values x_i: $[x_0, x_1, \ldots, x_{31}]$ as the private key.
2. Derive the public key by applying H to each x_i 256 times: $[H^{256}(x_0), H^{256}(x_1), \ldots, H^{256}(x_{31})]$.

Signing a Message:

1. Hash the message M to receive a 256-bit hash $H(M)$. Split $H(M)$ into 32 bytes n_i: $[n_0, n_1, \ldots, n_{31}]$.
2. For each $i \in [0, 31]$, apply n_i iterations of H to x_i to receive $H^{n_i}(x_i)$.
3. Publish the signature $S = [H^{n_0}(x_0), H^{n_1}(x_1), \ldots, H^{n_{31}}(x_{31})]$.

Verifying a Signature:

1. Given a signature S, let $S[i]$ refer to the i-th entry in the signature ($H^{n_i}(x_i)$). Let $PK[i]$ refer to the i-th entry in the public key ($H^{256}(x_i)$).
2. For each $i \in [0, 31]$, [ANSWER TO Q7.1].

Q7.1 (3 points) Let n_i be the i-th byte of $H(M)$, treated as an unsigned 8-bit integer.

Select the best option to fill in the blank from the signature verification protocol.

- ○ Let $T = H^{n_i}(x_i)$. Verify that $T = S[i]$.
- ○ Let $T = H^{256-n_i}(S[i])$. Verify that $T = PK[i]$.
- ○ Let $T = (H^{-1})^{256-n_i}(PK[i])$. Verify that $T = S[i]$.
- ○ Let $T = H^{256}(x_i)$. Verify that $T = PK[i]$.

Q7.2 (4 points) Which properties are *necessary conditions* for the signature scheme to be secure?

- ☐ $H, H^2, \ldots, H^{256}$ are one-way
- ☐ H is collision resistant
- ☐ H is secure against length-extension attacks
- ☐ The message never has a byte of all ones
- ☐ The output of H never has a byte of all zeroes
- ☐ None of the above

Q7.3 (6 points) Alice sends Bob a message M with signature S, generated with her private key. Mallory is eavesdropping on their conversation and learns (M, S). Mallory wishes to find a new message/signature pair (M', S') that verifies under Alice's public key PK.

Let n_i, n'_i be the i-th bytes, parsed as an 8-bit unsigned integer, for $H(M), H(M')$ respectively. What must be true **for all** $i \in [0, 31]$ for Mallory to succeed in forging a signature for M'? Select the most accurate option.

- ○ $n_i \leq n'_i$
- ○ $n_i = n'_i$
- ○ $n_i > n'_i$
- ○ $n_i \geq n'_i$
- ○ $n_i < n'_i$
- ○ $n'_i < PK[i]$

Assuming the message M' satisfies the correct condition, Mallory then calculates S'. For each $i \in [0, 31]$, give an expression for $S'[i]$, in terms of $n_i, n'_i, H, S[i], PK[i]$ (you do not need to use all those variables):

Q7.4 (2 points) Assuming H is a secure hash function, what is the approximate probability of the condition in the previous subpart being true for two randomly-selected messages M, M'?

- ○ 2^{-8}
- ○ 2^{-32}
- ○ 2^{-128}
- ○ 2^{-16}
- ○ 2^{-64}
- ○ 2^{-256}

Q7.5 (3 points) Select all options which would decrease the probability of the condition being true.

- ☐ Increasing the length of each x_i to be greater than 256 bits.
- ☐ Using 2 bytes for each n_i instead of 1 byte (e.g., n_0 would now be the first 2 bytes of $H(M)$ parsed as a 16-bit integer). Assume the public key values change to $H^{2^{16}}(x_i)$ accordingly.
- ☐ Using a hash function with a 512-bit output (assume that there would be 64 1-byte entries in S and PK rather than the original 32 entries accordingly).
- ☐ None of the above

Nothing on this page will affect your grade.

Post-Exam Activity

Help EvanBot out by drawing some toppings on their pancakes!

Comment Box

Congratulations for making it to the end of the exam! Feel free to leave any thoughts, comments, feedback, or doodles here: