Homework 3

1 Problem 1 (10 points)

For many block cipher encryption modes such as CBC mode, messages need to be a multiple of the block size. Messages that are not a multiple of the block size can still be encrypted, but need to be padded to a multiple of the block size. The padding moreover needs to be reversible so that the receiver can recover the original (unpadded) message when decrypting. For each of the following padding schemes, decide if the padding is reversible: that is, for any message, after padding to a multiple of the block length, it is possible to recover the message again. If the padding is reversible, explain how to recover the message and why recovery is guaranteed to work. If not, explain how it fails.

(a) **Null Padding**: Append 0's to the message until it is a multiple of the block length

(b) **Bit Padding, version 1:** Let N be the number of bits necessary to pad to a multiple of the block length. If $N > 0$, append 10^{N-1} — that is, a 1 followed by $N - 1$ 0's — to the message. If $N = 0$ (the message is already a multiple of the block length), do nothing.

(c) **Bit Padding, version 2:** This is the same as part (b), except that in the case $N = 0$, we append an entire block, set to 10^{B-1}, where B is the block length in bits.

(d) **PKCS7 Padding:** Assume the message is an integer number of bytes, but not an integer number of blocks. Let N be the number of *bytes* necessary to pad to a multiple of the block length. To ensure that $N > 0$, if the message is already a multiple of the block length, let N be the block length (in bytes). Now pad with N bytes, each byte set to the value N. For example, if $N = 3$, append 3 bytes to the message, each byte set to 00000011.

(e) PKCS7 padding, except that if the message is already a multiple of the block length, do not add any padding.

2 Problem 2 (20 points)

Consider CBC mode encryption, using PKCS7 padding from Problem 1d. We now will see a potential attack on CBC mode encryption using this padding. Suppose Alice is sending messages to a server, encrypted with CBC mode using PKCS7 padding. The server decrypts the CBC encryption, and then checks that the padding is correct: it verifies that there is some $N > 0$ such that the last N bytes of the plaintext (after decrypting, but before stripping the padding) are set to N. If correct, the server does nothing. If not, the server sends a "reject" message back to Alice indicating that the message was not well-formed.

Suppose an adversary Eve sits between Alice and the server. Eve can intercept messages from Alice, as well as send messages to the server, and read the response from the server. However, Eve does not know the secret key Alice and the server are using to communicate. Here is a sketch of Eve's attack:

- Eve intercepts a ciphertext c from Alice.
- Eve has a guess g for the last byte in the last full block of plaintext. In other words, if the message length is $\ell s + t$ bytes, where ℓ is the number of bytes per block and $t < \ell$, then Eve has a guess g for byte ℓs.
- Eve first strips off the last block of c, obtaining c'
- Next, Eve changes some portion of c', obtaining c''.
- Eve sends c'' to the server, and based on the server's response, learns whether or not g was a correct guess.

Answer the following:

(a) Fill in the details of the attack, namely decide how Eve computes c''. [Hint: you only need to modify one byte of c']

(b) Extend the attack to actually learn byte ℓs given no other information about the message, by making multiple queries to the server. [Hint: there are only 256 possible values for that byte]

(c) Under what circumstances will the attack in (a),(b) not be guaranteed to give the right answer? Can Eve tell if this situation arises?

(d) Explain how to still recover the byte, even if the attack in (a),(b) was inconclusive.

(e) Explain how to extend the attack to recover the entire message. This includes learning the first ℓs bytes, as well as the last t bytes.

(f) But wait! We said that CBC encryption was CPA secure, and yet we just showed how to recover the entire message. Explain why this is not a contradiction.

3 Problem 3 (20 points)

In class we saw that authenticated encryption implies CCA security. Show that the converse is *not* true: that is, show that CCA security does *not* imply authenticated encryption. To do so, give an example of an encryption scheme that is CCA secure, but which is not authenticated encryption. You may assume a secure block cipher, or any objected that is derivable from a secure block cipher (a CMA-secure MAC, an authenticated encryption scheme, etc).

4 Problem 4 (10 points)

In this problem, we will explore how to pad messages in order to make Merkle-Damgard collision resistant, even if the messages are different lengths. Let $h : \{0,1\}^{512} \to \{0,1\}^{256}$ be a compression function. We will apply the following padding:

- First, pad the message to a multiple of 256 bits by appending a string of the form $100\ldots00$.
- Suppose after the previous step, the message is of length $256 \times \ell$. Next, write down ℓ as a 256-bit number, and append ℓ to the message

For example, a string x of length 500 will get padded to $x10^{11}0^{254}10$, where 10^{11} is the padding from the first step, and $0^{254}10$ is the representation of $\ell = 2$ as a 256-bit number.

The actual hash function H applies the padding above to get a padded string whose length is a multiple of 256 bits, and then applies the Merkle-Damgard hash function.

(a) Show that any collision for H can be turned into a collision for h, even if the original collision consisted of messages of different lengths.

(b) Suppose we only apply the first step (appending $100\ldots00$), but do not append the length ℓ before applying Merkle-Damgard. Show that the collision resistance of h is *not* enough to demonstrate the security of H. In particular, explain how, for a poorly chosen IV, h can be collision resistant, but nevertheless it is possible to find collisions in H.

(c) What happens if we put the length ℓ at the *beginning*, instead of at the end?

5 Problem 5 (20 points)

One problem with Merkle-Damgard is that it is sequential, and cannot take advantage of parallelism to speed up the computation.

An alternative hash function is described as follows.

Let $h : \{0,1\}^{512} \to \{0,1\}^{256}$ be a compression function. Define $H_i : \{0,1\}^{256 \times 2^i} \to \{0,1\}^{256}$ as follows:

- $H_1 = h$
- H_{i+1} does the following. On input a $256 \times 2^{i+1}$-bit message x, write x as $x_0 x_1 \ldots x_{2^i - 1}$, where each x_j is a 512-bit string. Let $y_j = h(x_j)$ for all j, and let $y = y_0 y_1 \ldots y_{2^i - 1}$. Output $H_i(y)$ as the hash.

In other words, H_i is defined by a binary tree containing 2^i leaves, which contain the various 256-bit blocks of x. The value at each node is set to be the hash under h of the concatenation of its two children. The output is the value of the root.

Given sufficiently many processors, one can compute H_i in about i time steps.

(a) Show how, given a collision for H_i, you can construct a collision for h

(b) The above hash functions only work for messages consisting of an number of blocks that is a power of 2. Explain how to make the hash function work for arbitrary length messages. Your hash function should use essentially the same number of hashes as Merkle-Damgard would use on the same message (it can be, say, an additive logarithm more, but shouldn't be double). Your hash function should also be collision resistant even for messages of different lengths