Tilda Jansson

Walkthroughs for CryptoPals Challenges

Set 1

1.
This code manually converts hexadecimal characters to base64.
The function begins by converting the hexadecimal input to a binary string, and since Base64 is meant to encode bytes (8 bit) we pad the binary string with leading zeros on the left side to make its length a multiple of 8 bits.

Next step is to divide the binary data into 6-bit chunks (since each Base64 character represents 6 bits). If the length of the binary string is not a multiple of 6 we append additional zeros to the end of the binary string to make sure it can be evenly divided into 6-bit chunks.

Then we split the binary string into 6-bit chunks and look up the corresponding Base64 character for each.

The encoded Base64 data must always be a multiple of 4 characters since Base64 is 6-bit encoding and the decoded bit string is divided into 8-bit bytes, which means that 4 Base64 encoded characters represents 3 8-bit bytes of unencoded data. I.e. the unencoded binary data input must be a multiple of 24 bits since LCM(8,6) = 24 for the encoding to be compatible with the decoding, otherwise padding is needed.

If the encoded Base64 string is not a multiple of 4 characters, padded characters ('=') are appended to the string until it is a multiple of 4. The padding indicates how many trailing bits from the decoded bit string should be removed. Each time a '=' character is encountered, 2 trailing bits are removed.

If there are two '=' characters in the encoded string it means that the last 4 characters decode to 1 byte. If there is one '=', it means that the last 4 characters decode to 2 bytes.

In the decoding we convert each character (except the trailing occurrences of '=') into their corresponding 6-bit representation, and then remove 2 bits for each '='.

2.
This 'xor' function applies the bitwise XOR operation between corresponding bytes of the two input buffers of equal length. It also checks that the lengths of the two buffers are equal, if they are not, the function raises a 'ValueError'. The function returns a bytes object containing the result of the XOR operation performed across the buffers.

3.
To find the key (a single-byte) that the message has been XOR'd against, we try all possible keys (256 possibilities for an 8-bit key), xor the message against that key and score the resulting plaintext based on character frequency. Decrypting is the same as xoring the ciphertext against the key since XORing twice by the same value gives back the original value.

For each possible key it creates an extended key of the same length as the cipher text by repeating the key byte, XORs this extended key with the cipher text and then scores this plaintext.

Each plaintext is scored for how similar it is to typical English text using the ‘score_text’ function which uses a dictionary that maps characters to their relative frequencies in English text to calculate how English-like a text is. It looks up the corresponding frequency value for each character of the plaintext and the sum of all these frequencies is the total score.

Once all possible keys have been tried, the function returns the decrypted text with the highest score and its corresponding key. A higher score suggests a closer match to the expected frequencies.

4.
This function finds the line in a given file that has been encrypted with a single-character XOR and decrypts it. It assumes that the most "English-like" decrypted line (determined by character frequency scoring) is the correct decryption.

It iterates through each line of the file and converts the hex-encoded string to bytes. For each line, it tries all possible keys and returns the best one based on the highest score for the decrypted text (it reuses the function single_byte_xor_cipher from challenge 3). Then we have the best key, the decrypted text and the score for that line which we compare to the best overall score for all lines. It keeps track of the decrypted text that best matches typical English text based on the highest score achieved, as well as its key and score, and updates the "best" variables (text, key, score) when it finds a line with a higher score than previously recorded.

In the end, the function prints the decrypted line number, the key value and the decrypted text.

5.
The function encrypts plaintext using a repeating-key XOR. First it extends the key to match the length of the plaintext which is done by repeating the key as many times as needed to ensure it is at least as long as the plaintext, and then trimming it to get the exact length of the plaintext. Then it xors the extended key with the plaintext using the function from challenge 2 to get the encrypted data. These encrypted bytes are then returned.

6.
In order to determine the correct key used for the repeated xor encryption we define four functions.
The function ‘possible_key_sizes’ calculates the average hamming distance for each key size [2,40] and ranks them in ascending order since the key sizes with shorter hamming distance are more likely to be accurate (the distance is the number of differing bits determined by xoring byte buffers and counting the number of 1 bits in the result).

It calculates the average hamming distance for each key size by comparing a maximum number of pairs of blocks of the key size. For each pair it calculates the hamming distance and normalizes the value by dividing by the key size, this value is then summed up for all the pairs and then it gets divided by the number of pairs to get the average distance for that key size. When we have iterated through all potential key sizes we have a list of tuples of type (avg_distance, keysize). This is then sorted in ascending order according to distance.

The function ‘key_for_keysize’ is then called on this list of sorted key sizes. This function tries to find the most probable key for each key size. It iterates through the key sizes and for each byte-position in the key size, it slices with the step of the key size. Thereby gathering all bytes that would be XORed with the same byte of the key in a block, as if the key was applied in a repeating fashion.

It then uses the function ‘single_byte_xor_cipher’ from challenge 3 on each block to determine the corresponding key-byte. When we have all key-bytes for a key size we add it to a list of possible keys. The function then returns this list of possible keys.

The final step is to determine the correct key out of the possible ones. We do this by calling ‘find_correct_repeating_xor_key’ on the list. This function iterates through all the possible keys, and for each key it extends the key by repeating it to get the same length as the ciphertext. The extended key and the ciphertext are then xored to decrypt the text and then a score is calculated based on how similar it is to typical english text based on character frequency. We keep track of the best score and the corresponding decrypted text and key, while iterating through the possible keys. The most probable key and its decrypted text are then returned.

7.
The goal of this challenge is to decrypt ciphertext encrypted via AES-128 in ECB mode under a specified key. However, since the CBC mode of AES will be used in later challenges I implemented a class called ‘AESCipher’ including both modes. The ECB mode decryption in this class works as follows:

The ECB mode decrypts each 16 byte block independently using the same 16 byte key for each block. It first checks that the ciphertext is a multiple of the block size. The ‘_aes_ecb_decrypt’ function decrypts the provided bytes using AES in ECB mode by creating a cipher object with the specified key and mode, and then applying the cipher's decrypt method to the data. When all blocks have been decrypted we unpad and return the plaintext.

8.
To detect which ciphertext in the file that has been encrypted with ECB we use the fact that ECB encrypts each block using the same key and that it is deterministic, i.e. using AES with the same key and same plaintext always gives the same output. This means that if there are two identical blocks in the plaintext, there will be two identical blocks in the ciphertext.

Therefore, we split each ciphertext in the file into blocks of 16 bytes and check if the number of blocks in the set of the blocks is the same as the total number of blocks. If the numbers differ this means that there are duplicate blocks in the ciphertext, indicating that ECB mode was used. The line number of the ciphertext and the ciphertext are returned.

Set 2

9.

In this challenge we implemented PKCS#7 padding and unpadding. These functions are used for handling plaintexts that are not naturally a multiple of the block size, which is a requirement when using AES. The 'pad' function adds padding to the data so that its length becomes a multiple of block_size. In the case that the plaintext is actually a multiple of the blocksize, an additional block of padding is added (otherwise it would be difficult to guess the padding length when removing it).

The function starts by calculating the number of padding bytes needed to fill the last block. It then uses that value to craft the padding since the padding is a series of bytes, each of which is the same as the total number of padding bytes required. These padding bytes are then appended to the original data which is returned.

On the other hand, the 'unpad' function removes padding from data that was previously padded to align with the block size. This is used in the final step in the decryption process, after the actual decryption of the data. It starts by checking the number of padded bytes by looking at the last byte of the padded data. It then verifies that the padding is correct by checking if the padded bytes all have the same value as the number of padded bytes. If the padding does not match this pattern, it raises a ValueError indicating invalid padding. If the padding is valid, it returns the data with the padding removed.

10.
In the class AESCipher we implemented the methods for CBC encryption and decryption, using ECB mode operations as the core. The '_enprypt_cbc' function first pads the plaintext to make sure it is a multiple of the block size. It then XORs the IV with the first block, encrypts the results and adds it to the ciphertext. For subsequent blocks, each block is XORed with the ciphertext of the previous block before being encrypted and added to the ciphertext. This chaining ensures that the same plaintext block will produce different ciphertext blocks depending on the preceding block's ciphertext, enhancing security.

For the decryption process, '_decrypt_cbc' first splits the ciphertext into blocks of 16 bytes. Then each block is decrypted using AES in ECB mode. The decrypted block is then XORed with the previous ciphertext block (IV for the first block) to recover the original plaintext. This process reverses the encryption steps, with the XOR operation canceling out the one performed during encryption, due to the XOR operation's properties. The last step is to unpad the plaintext.

While CBC mode involves chaining blocks together, the actual cryptographic operation on each block is still just AES encryption or decryption in ECB mode.

11.
This code implements an encryption oracle that randomly encrypts data using either AES in ECB or CBC mode. It also includes a function to detect which encryption mode was used based on analyzing the encrypted output.

Since more oracles are needed in later challenges an interface is created with a placeholder ‘encrypt’ method. The ‘randomEncryptionOracle’ extends the interface and implements the ‘encrypt’ method. The method first adds random padding to both the beginning and end of the plaintext. This random padding varies in length between 5 to 10 bytes. It then randomly chooses between ECB and CBC mode for encryption. For CBC mode, an IV is also generated randomly.

The chosen mode used is recorded in the field ‘used_modes’ in order to compare the true modes used to the result of the detection function. The ciphertext is returned.

The ‘detect_ECB_CBC_mode’ function tries to determine if the encryption was done using ECB or CBC mode by looking for duplicate blocks in the ciphertext. It encrypts repeated characters (48 'A's) using the oracle and checks for repeating blocks in the ciphertext. The presence of duplicate blocks indicates ECB mode; otherwise, CBC mode is assumed. It returns the guessed mode.

In the main function, the ‘detect_ECB_CBC_mode’ function is run 100 times with each guessed mode added to a list. That list is then compared to the ‘used_modes’ list of the oracle which contains the actual modes used. It prints the number of times the detected mode matches the actual mode, testing the accuracy of the ECB/CBC detection logic.

12.
The goal of this challenge is to implement the "byte-at-a-time ECB decryption" against an encryption oracle that uses AES in ECB mode, and decrypt an unknown string that is appended to plaintext before encryption.

First the class ‘ECB_EncryptionOracle’ which extends the previously mentioned oracle interface. The class Initializes with a random AES key, an AES cipher instance, and an unknown string that is base64-decoded. The encrypt method appends the unknown string to a given plaintext and then encrypts the combined text using AES in ECB mode.

The ‘find_block_size’ function determines the block size used by the oracle by encrypting increasing lengths by increasing the number of the character ‘A’ (or any ASCII character) in the plaintext until the length of the ciphertext changes. The difference in length is the block size. This is because we know that a complete new padded block is added when the plaintext is a multiple of the blocksize.

The ‘find_payload_length’ function first calculates the length of the ciphertext when no plaintext is added, just the unknown text. Then, it calculates how much padding was added to the unknown text by increasing the number of characters in the plaintext and seeing when the length of the ciphertext changes. It then subtracts the number of added characters from the initial length of the ciphertext when no plaintext was added. That is the length of the unknown text.

The ‘decrypt_byte_by_byte’ function implements the byte-at-a-time ECB decryption attack.
For each unknown byte it creates a byte string of 'A's that is one/several byte(s) short of the block size, depending on the number of discovered bytes, such that the last byte of the discovered bytes added to the series of ‘A’ is unknown. By comparing the encrypted output of this combination and the combination plus a guessed byte, we can identify the correct byte when the ciphertexts match up to and including the target byte. It aligns the target byte to be the last byte in a block, then iterates through all possible byte values to find a match in the ciphertext. This process is repeated byte-by-byte to decrypt the entire unknown string which gets returned.

13.
We started off by defining a class ‘UserProfileECB’ that initializes a randomly generated 16-byte key and creates an instance of AESCipher with this key for encryption and decryption operations. THe class includes the following methods:

The parse method takes an encoded string of key-value pairs and decodes it into a dictionary. It splits the string by the '&' character to separate the pairs and then further splits each pair by ‘=’ to get keys and values. It constructs and returns a dictionary from these keys and values.

The profile_for method encodes a user profile for a given email address. The ‘uid’ and ‘role’ values are fixed. It checks the email address to ensure it does not contain ‘&’ or ‘=’ characters. It then encodes the profile into a byte string in the format key=value&key=value... for all profile attributes.

The ‘encode_and_encrypt_profile’ method takes an email address as input, encodes the corresponding user profile into the key=value&key=value... format, and then encrypts this byte string using AES in ECB mode.

The decrypt_and_parse_profile method decrypts an encrypted profile back into its encoded form using the same AES key and ECB mode. It then parses the decrypted byte string back into a dictionary using the parse method.

The attack:
The goal of the attack on this profile manager is to craft an encrypted encoding of a user profile where the role is set to “admin”. This is done by only using the profile_for() and manipulating the ciphertexts themselves.

We start by creating an email address that when used to generate an encoded user profile, aligns the “...role=” in one block and the “user” part into separate AES blocks. This is done by padding the email field with a specific number of ‘A’ characters (13 in this case) to manipulate the placement of subsequent fields in the profile.

We then encrypt this encoded profile but remove the last block, since that block contains “user + padding”, which is what we want to replace. Next step is to craft a new block that only consists of ‘admin’ + padding, using the ‘pad’ function. Then we try to fit this admin block in a new profile encoding so it appears in a separate block (here we place it in the second block). This is achieved by adjusting the number of ‘A’s in an email to ensure “admin” starts at the beginning of a new block, we use 10 ‘A’s. Then this profile encoding is encrypted and the second block is extracted (the admin block).

The final step is to concatenate the first part of the encrypted email, the one whose encoded profile ends with ‘...role=’, with the extracted encrypted padded admin block. This yields ‘role=admin’ when decrypted and parsed.

14.
In this version of the byte-at-a-time ECB decryption attack we create a new oracle class that extends the oracle interface from #12. This oracle encrypts a string with some random prefix and an unknown target postfix. The random prefix is generated once at the instantiation of the oracle, and stays the same across all calls to the oracle.

We start off by calculating the block size, the initial length consisting of just the prefix and unknown postfix (+ padding), as well as the length of the potential padding for that initial length. The block size is calculated in the same way as in previous challenges, i.e. by observing the increase in ciphertext length as the plaintext length is incrementally increased. The amount of plaintext needed for the change in length is the padding used in initial encryption where no user plaintext was added.

The next step involves finding the length of the prefix. The function determines the length of the random prefix by inserting ‘A’s and observing when two consecutive identical blocks appear in the ciphertext. Then we know which two blocks are filled with ‘A’s, as well as the total number of ‘A’s added. To calculate the prefix length we subtract 2 block sizes from the total number of ‘A’s and then take the index of the first consecutive block multiplied by the block size minus the rest of the ‘A’s.

Now that we know the initial length (prefix + unknown postfix + padding), the length of the padding for that initial length, and the length of the prefix, we can calculate the length of the unknown postfix. Postfix length = (prefix + unknown postfix + padding) - (calculated padding) - (prefix length).

Final step is to decrypt byte by byte in the same manner as in #12. The main difference is that we now pad the prefix with the character ‘A’ to a full block size, which allows us to use basically the same code as in #12 with minor modifications to be consistent with the prefix and its padding.

15.
The ‘unpad’ function removes PKCS#7 padding from data. Initially, it checks whether the length of the data is a multiple of the block size and not an empty byte string. If not, it raises a ValueError indicating that the input data is not properly padded or is empty.

It then retrieves the last byte of the input data, which indicates the number of padding bytes added. If this value is greater than the block size or less than 1, the function raises a ValueError, as this would be invalid padding. Next, it verifies that all padding bytes are the same and match the expected value (the last byte of data). If the check fails, it again raises a ValueError. Finally, if the padding is valid, it returns data with the padding removed.

16.
We create an oracle class with an encrypt and a decryption method. In the 'encrypt' we sanitize the input by escaping semicolons (;) and equals signs (=) by prefixing them with a backslash (\). It then prepends and appends fixed byte strings to the plaintext and encrypts the result using AES in CBC mode with randomly generated key and IV.

The 'decrypt_check_admin' method decrypts a given ciphertext using the same key and IV, then checks if the decrypted plaintext contains the byte string ";admin=true;". If it is found 'True' is returned, otherwise 'False' is returned.

CBC bit flipping attack:
We know that if we flip 1-bit in a block of the ciphertext that block will look scrambled when decrypted, but more importantly, that specific bit flip in the ciphertext will directly affect the corresponding bit in the plaintext of the next block, due to the XOR operation with the modified ciphertext block during decryption.

Since we also know that the prefix is exactly 2 blocks (making it easy to flip the right bits) and that the postfix begins with a ';', which means it is enough to plug in ";admin=true".

We want to flip the bits in the second block of the prefix ciphertext, ctxt1, in order to modify the first 11 bytes of the third block to get ";admin=true". We do this by encrypting 11 'X' characters, which yields that the encryption of the third block is ctxt2 = enc(ctxt1 XOR ptxt2), where ctxt1 is the ciphertext of the second block and ptxt2 is the third block plaintext starting with our 11 'X's.

The next step is to bitflip the ciphertext of the second block, so we get ctxt1 = (ctxt1 XOR 'X'*11 XOR ";admin=true"). In the decryption process when we come to the third block, the block ctxt2 = enc(ctxt1 XOR ptxt2) is first decrypted and then XORed with the bitflipped ctxt1 block. This gives:

(ctxt1 XOR ptxt2) XOR (ctxt1 XOR 'X'*11 XOR ";admin=true") = ptxt2 XOR 'X'*11 XOR ";admin=true"

Since ptxt2 is the third block plaintext starting with our 'X'*11, the 'X's cancel out each other, and we are left with the desired ";admin=true" in the decrypted plaintext.

To answer the question "Why does CBC mode have this property?" (regarding error propagation):
It is because of the chaining mechanism of CBC. If a bit is flipped in a ciphertext block the same bit in the plaintext of the next block is also changed since the plaintext block is XORed with the previous erroneous ciphertext block.