Part 1

Question 1

In the file names "plaintext1" is the resulting plaintext from the decrypted file "ciphertext1". They key is 7 characters long in the hex order "{0x50},{0x2F},{0x08},{0x7C},{0x5F},{0x30},{0x00}".

Question 2

From the ciphertext, we used Kasiski's examination to correctly find the length of the key. We found the distances between certain groups of lettering and then calculating the factors based on the length of distance. E.g. the grouping "76 b5 6e" appears multiple times in the ciphertext and the distances between some of them are is at least 21. Combining this with other character groupings we can calculate the common factor between all of them to be 7. Therefore the key length is 7.

Once the key length was found, we then split up the ciphertext into 7 "bins" such that each bin had the first character of the first row and then the remaining characters following by multiples of 7.

Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6	Bin 7
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21

We then take the first and second character from the first bin and calculate all possible keys for each and pull out the keys that exist in both lists. This creates a "master list" of possible keys for this bin. Take the third key and calculate all possible keys and compare to the master list for similarities and take only the ones that exist in both. Do this until we only have possible key left that will decrypt the entire bin. Repeat for the rest of the bins. This will give us the full key.

Question 3

By using the method in question 2, we can determine whether we have a possible key by outputting the calculated key afterwards. The program will then immediately start decrypting the ciphertext using the newly found key and start outputting it into the output file provided during execution. We can then open the output file to determine if the whole file is readable as the plaintext should be printable ASCII characters.

Question 4

See README.md or REPORT.pdf for compilation and execution.

Question 5

If the plaintext file had been first compressed then the strategy would have been similar to part 2. Since zip files also have file header information in it that will determine what the file should be, we can use this information to find out how large the uncompressed file is, what the name of it is, etc. The only place that we could use Kasiski's examination is in the last part of the file where we can calculate the the distances of the file header.

Reference

https://en.wikipedia.org/wiki/Zip_(file_format)

https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher#Kasiski_examination

Part 2

Question 1

The decrypted file is given as "output.jpg". The key is 23 characters long, given by these hexadecimal bytes:

{0x35, 0x33, 0x2E, 0x35, 0x30, 0x33, 0x35, 0x36, 0x33, 0x4E, 0x2C, 0x2D, 0x31, 0x31, 0x33, 0x2E, 0x35, 0x32, 0x38, 0x38, 0x39, 0x34, 0x57}

Question 2

Similar to part 1, we used Kasiski's examination to see if we could find the length of the key. The distances between groupings of characters was siginificantly larger than in part 1, but after enough evaluation we found that 23 occurred frequently as a common factor. We then used the code from part 1 to evaluate ciphertext2, splitting it instead into 23 "bins", but the evaluation was unsuccessful and the code from part 1 was unable to find a key.

Here we realized we needed to deviate from our solution to part 1. We knew that being able to figure out what file type the plaintext is would be able to help us, so we looked at a range of different file signatures. Seeing as how the header for any file is a known plaintext, we used that, the ciphertext, and the given mapping to determine what the first bytes of the key would have to be if the plaintext were each file type.

Knowing the plaintext is supposed to be a common file type, we looked at file signatures for DOCX, DOC, PDF, PS, PNG, and JPG files and found what the keys would have to be if the plaintext were any of those. Seeing as how we know the key is limited in this part to printable ASCII characters (0x20 to 0x7f), we were able to immediately eliminate all of the file types we had picked but several JPG ones, as the partial keys for their headers all had at least one non-printable ASCII character in them. From the reference site we found six common options for the first four bytes of a JPG, three of which we were able to eliminate as their keys had a non-printable ASCII character in them.

Left with either a Samsung D807 JPEG file, Standard JPEG/Exif file, or Still Picture Interchange File Format (SPIFF) as potential headers, we decided to examine the ciphertext using each of the three combinations of initial four key bytes. We went through the first 10,000 ciphertext bytes decrypting every four with a potential key. In this way bytes 0-3 would be decrypted, then 1-4, then 2-5, and so on. The idea is that if we do in fact have the initial four bytes of the key, then at some point a decryption of four bytes should make sense and we would have a fairly good idea about the key length.

We looked at the hexadecimal of several JPGs we had to get a feel for the file structure, particularly to note anything that stood out. Two that really stood out early on in each were a timestamp and the occurrence of a bunch of 2s all in sequence. We ran the decryption with the three different JPG formats, and with the Standard JPEG/Exif file we got promising results. At the 575th byte we noticed the sequence of 4 bytes looked like it could be from a timestamp, so we then doubled that number and checked the 1150th byte to find a sequence of four 2s. We realized then that 575 was divisible by 23, so this with our findings from Kasiski's examination early on led us to believe the key length had to be 23 and that we had found the file type.

Having a strong hunch about what kind of JPG we were dealing with we looked at the full file signature which is 11 bytes long (where the 5th and 6th bytes are unknown). We then decrypted the whole ciphertext2 using a 23 byte key with the 9 bytes we knew filled in (and the others left as garbage). Taking a look at the hex dump of the decryption we could see in greater detail where the sequence of 2s was, and so knowing that the bytes we didn't have the key for in between two groupings of 2s should definitely be 2s, we had a known plaintext and ciphertext. We were able to then find the missing key bytes, and with the completed key we then decrypted the whole ciphertext.

Question 3

We mostly calculated the key manually, using code to help with mapping and decryption. Once we had what we felt was the key, we gave it to the code to decrypt ciphertext2 with and output the result to "output.jpg". As we got a clear picture from the result we knew we had the right key and the right file format. The pattern recognition parts of finding and verifying the key would have been far too difficult to try to code and automate in any effective way, so we stuck with manually doing it for the most part, only using code to help with the simple but tedious tasks of mapping and decryption.

Question 4

See README.md in lab1 for compilation and execution.

Reference

http://www.garykessler.net/library/file_sigs.html

Part 3

Question 1

The plaintext file consisting of 10 repetitions of the string `01234567`
is named as `plaintext`.

Question 2

The encryption key used for all the cipher modes of **DES** is `nosalt`; and
the corresponding encrypted outputs for the 4 modes of **DES** are named as
`cipherecb.enc`, `ciphercbc.enc`, `ciphercfb.enc`, and `cipherofb.enc`,
respectively.

Question 3

Size Differences

ECB

The encrypted output `cipherecb.enc` is 8 bytes larger than the original
`plaintext` due to the extra 8-byte-padding added by `openssl` using the
PKCS5 scheme, which is enabled by default for block ciphers.
Source quoted from the `enc` manual page of `openssl`:
All the block ciphers normally use PKCS#5 padding also known as standard
block
padding: this allows a rudimentary integrity or password check to be performed.
If padding is disabled then the input data must be a multiple of the cipher
block length."

CBC

The encrypted output `ciphercbc.enc` is 8 bytes larger than the original `plaintext` due to the extra 8-byte-padding added by `openssl` using the PKCS5 scheme, similar to the reason given for the **ECB** mode.

CFB

The encrypted output `ciphercfb.enc` is of the same size as the original `plaintext` file (80 bytes) because the final size of output ciphertext blocks is the same as the plaintext and there is no padding bytes added by `openssl`.

OFB

The encrypted output `cipherofb.enc` is of the same size as the original `plaintext` file (80 bytes) because the final size of output ciphertext blocks is the same as the plaintext and there is no padding bytes added by `openssl`.

Content Differences

ECB

The encrypted output `cipherecb.enc` has a repeating pattern `e0b1 304d a28a fd3b`, which corresponds to the repeating pattern of the original file `3031 3233 3435 3637`.

CBC, CFB, OFB

The encrypted output files `ciphercbc.enc`, `ciphercfb.enc`, and `cipherofb.enc` does not have any repeating pattern possibly because `openssl` makes changes to the IV (initialization vector, it is not explicitly specifed as a commnad line arguemnt), which results in the same repeating plaintext **01234567** being enciphered to a different output.

Question 4

The "error" version of each file is named `cipherecberror.enc`, `ciphercbcerror.enc`, `ciphercfberror.enc`, `cipherofberror.enc`, respectively.

Question 5

The decrypted output files are named as `plaintextecb`, `plaintextcbc`, `plaintextcfb`, and `plaintextofb`, respectively.

ECB

After decryption, only the *third* **01234567** block is corrupted because blocks are enciphered **independently** of other blocks in the **ECB** mode; in this case the 19th corrupted byte belongs to the *third* 64-bit (8-byte) block, only this block is affected after decipherment.

CBC

After decryption, the *third* and *fourth* **01234567** blocks are both corrupted: the difference is that the 19th byte error propagates to the entire *third* block but only the third byte in the *fourth* block (*2*) is affected. The reason for this is that the 19th byte is the *third* byte within the *third* (64-bit) block, in **CBC** mode a byte error in ciphertext block j affects decipherment of blocks j and j + 1; however, the extra *xor* (exclusive or) operation during decryption between cipher block j and j + 1 (in the case the *third* and *fourth* block) would preserve the exact location of the byte error in j + 1 plaintext block (in this case the *third* byte in the *fourth* block).

CFB

After decryption, the *third* and *fourth* **01234567** blocks are both corrupted: this time only the 19th byte (corrupted byte) is affcted in the *third* block but the error also propagates to the entire *fourth* block.
The reason for this similar scenario with respect to the **CBC** case is that the decryption of **CFB** mode is almost identical to the **CBC** encryption performed in reverse: after the *xor* operation for decryption the byte error is preserved in the exact location in block j (the *third* block in this case) but propagates to the whole j + 1 block (the *fourth* block).

OFB

After decryption, only the 19th byte (corrupted byte) is affected because the **OFB** mode does not have chaining dependencies among blocks, and there is a final *xor* (exclusive or) operation involved to decipher the plaintext from the ciphertext. In this case the original encrypted 19th byte is **0x3e**, the bitmask used to corrupt it is **0x5f**, so to obtain the correct plaintext, an extra *xor* operation needs to be performed between the corrupted plaintext byte **0x6d** and the bitmask used previously.

Question 6

The modified file is named as `ciphertext3.mod`.

Reference

- `man 1 enc`
- Chapter 7 of the Handbook of Applied Cryptography

Division of Workload

The submitted version of program for part 1 was written by Stephen Arychuk, he
also wrote the whole report for it and did the entire `openssl` related part of part 3; the program submitted for part 2 is entirely done by Stuart Bildfell; he wrote the whole report for part 2 as well; the report for part 3 was written by Jiahui Xie; he also helped with the file signature portion of part 2 and wrote another version of program for part 1 (without the actual password cracking part).