Problem Set 2

This problem set is due **at 8:00pm** on **Monday, October 21, 2024**.

- The TAs will provide a detailed document describing how you should submit your PDF and code on Google Classroom (we may use gradescope for some things). Make sure you read it! We suggest that you perform a trial submission prior to the deadline to make sure that everything works for you – you can overwrite that submission with a new one up to the deadline.
- We require that written solutions are submitted as a PDF file, **typeset on LaTeX**, using the template available on Google Classroom. You must **show your work** for written solutions. Each solution should start on a new page.
- We will occasionally ask you to "give an algorithm" to solve a problem. Your write-up should take the form of a short essay. Start by defining the problem you are solving and stating what your results are. Then provide: (a) a description of the algorithm in English and, if helpful, pseudo-code; (b) a proof sketch for the correctness of the algorithm; and (c) an analysis of the running time.
- We will give full credit **only** for correct solutions that are described clearly and convincingly.

Background

The goal of this problem is to explore the method of brute force attack on a cryptosystem to get a better feeling for what it can and cannot do. This problem was originally created by Mike Fischer, with edits from Ewa Syta and Debayan Gupta.

Our friend, Happy Hacker, has ignored my advice and decided to build his own cryptosystem, which he calls SnakeOil. He started with the botan (botan.randombit.net) implementation of AES-128 in CBC mode, but he decided to add his own padding and key management routines.

Padding Botan AES-128/CBC defaults to using PKCS7 byte padding. However, Happy was afraid that the redundancy it adds might make the code easier to crack since most incorrect keys will give a "decoding error" rather than a plausible-looking decryption. Instead, Happy decided to simply pad out the last incomplete block with zeros. For decoding, any trailing zero bytes in the last block are simply discarded. This works fine for text files, since they generally do not contain the zero (NUL/NULL) byte anyway, and it eliminates the possibility of decoding exceptions.

Key Management Happy didn't think 128-bit keys were long enough, so he came up with a clever scheme for extending the key space. He first generates a file of 100 random 128-bit strings called *key shares*. He then computes a 128-bit master key to be used with AES. The master key is specified by two indices $0 \leq idx1 < idx2 < 100$. The *master key* is simply the exclusive-or of the two key shares with indices $idx1$ and $idx2$, respectively.

Happy was pleased with his scheme and told his friends about all of its features.

1. It has a 12,800 bit key, which makes it far safer than AES's measly 128-bit key.
2. Happy reasons that it is now safe for him to store the key shares file on his hard disk since the file does not contain the master key and in fact is nothing more than a file of random numbers.
3. The only thing that Happy and his communication partners have to remember is the pair of key indices, both of which are numbers between 0 and 99. This is no more difficult than remembering the PIN for your bank ATM card.

Unfortunately for Happy, he forgot the correct pair of key indices and so was unable to decrypt a message he received from Alice. Clever Charlie told him not to worry. His system was easily compromised by anyone with access to the key share file. A program could be written to brute force the possible key pairs, and the computer could be decrypting Alice's message while they went out for pizza.

Sure enough, when they returned, Alice's message was displayed on the screen.

A Brute Force Attack

For Happy's problem, a brute force attack decrypts Alice's ciphertext c for every possible key index pair $(k_i, k_j), 0 \leq i < j < 100$ in Happy's key file until the correct pair is found. The principal difficulty is recognizing the correct decryption once it has been found. How can one distinguish the correct decryption from the wrong one?

In general one cannot, but if some messages are more likely than others, then some decryptions of c will "look more like a valid message" than others. The attack chooses the decryption m' of c that looks most like a valid message and guesses that it is the real message m. It also guesses that the key index pair $k' = (k_i, k_j)$ that produced it is the real pair.

The guess m' and k' will not always be correct. If they are, we say the attack *succeeds in breaking the cryptosystem*. Otherwise, the attack fails. We are interested in exploring how well this attack can be made to work in practice.

Letter Frequencies

Clever's method for choosing m' relies on the letter frequencies of English text. Assume you have a large corpus of representative English text, so that Alice's message is likely to have a similar letter frequency distribution. Assume further that a decryption of c using a randomly chosen incorrect index pair k' yields a random-looking string m' with more or less uniform distribution of the letters. If c is sufficiently long, then very likely the correct decryption m is the only one whose letter frequency distribution closely resembles that of the corpus.

Measuring Similarity

For each octet (8-bit byte) b, let $f(b)$ be the frequency of occurrence of b in a corpus of English text, and let $r = \sum_b f(b)$ be the total size of the corpus. Then the normalized frequency $p(b) = f(b)/r$ is a close estimate of the probability that an arbitrary byte in a random piece of text is equal to b.

Similarly, let $q(b)$ be the normalized frequency of b in the message. We measure the (dis)similarity of p and q by the sum of the squares of the difference at each byte of their normalized frequencies. That is, we define

$$divergence(p, q) = \sum_b (p(b) - q(b))^2$$

We apply this measure by computing the divergence between each possible decryption of c and the corpus and choosing the decryption (and corresponding key) that minimizes the divergence.

Questions

All work should be submitted electronically as per the TA's instructions. The written solutions should be in PDF form, using the LaTeX template provided. You will find a C++ implementation of SnakeOil in the `src` folder. You will also find a partially-written brute force key analyzer, complete with routines for reading and normalizing frequency tables. However, it is missing two important functions: `guessKey()` and `divergence()`.

Problem 2-1. Security Analysis [30 points]

Write a critique of SnakeOil from both a usability and a security standpoint. What is good and bad about the cryptosystem, and what is good and bad about this particular implementation? What is the effective key length of SnakeOil? Evaluate each of Happy's claims above for the superiority of SnakeOil over AES-128.

Do not limit yourself to answering just these particular questions. Rather, feel free to comment on all aspects of a cryptosystem that impact its usability and security in a particular environment. Your answer should reflect what you have learned in the course so far about security in general and what it means for a cryptosystem to be secure.

Problem 2-2. Coding [40 points]

You should write the two missing functions in the brute force key analyzer. Compile your program and test it on the file `sample.enc`. This should not take more than around 50 lines of C++ code, but of course it has to work in the context of the rest of the program, so you will need to spend some time reading the rest of my code in order to see how one invokes the AES primitives in order to decrypt a file with a particular master key.

I realize that some of you might not be very familiar with C++, so please feel free to ask for help, and take your time!

Problem 2-3. Experiments [30 points]

You will find several files in the `data` folder. Files ending in `.dat` are frequency tables. Files ending in `.enc` are ciphertexts. The file `keyshares` is the key share file that was used in all of the encryptions. All ciphertexts are valid encryptions of English-language text files, but not all are easy to decipher.

You should run your brute force key finder with every frequency table and every ciphertext file. For each, report the key indices that the program found, the decryption produced, and whether or not the program succeeded in finding the correct key.

Next, you should analyze your results and try to draw conclusions about the effects of the frequency table and the length of the ciphertext on the effectiveness of the attack. Based on the insights gained, construct a 40 character "message" that cannot be cracked using any of the furnished frequency tables. (Your message does not have to be real English text, but it must consist of printable ASCII characters.)