CPSC 418 / MATH 318 Introduction to Cryptography

ASSIGNMENT 1

Due: **Wednesday, Feb. 10, 2021** at **11:55 PM** Total marks: **100**

Prior to submission, be sure to familiarize yourself thoroughly with the assignment **Policies and Guidelines** as well as the **Specifications and Submission Procedure** as detailed on the assignments course webpage

http://people.ucalgary.ca/~rscheidl/crypto/assignments.html.

Assignments that don't follow these instructions will incur penalties of varying degree, up to a score of zero.

Problems 1-4 are worth 62 marks and are required for both CPSC 418 and MATH 318 students.

Problems 5 and 6 are worth 38 marks and are for MATH 318 students only; CPSC 418 may do these for limited extra credit.

Problems 7 and 8 are worth 38 marks and are for CPSC 418 students only; MATH 318 may do these for limited credit.

The bonus credit policy can also be found under the link above.

Written Problems for CPSC 418 and MATH 318

Problem 1 — Password length and entropy (20 marks)

ASCII, short for *American Standard Code for Information Interchange*, is a method for encoding any character into a string of 7 bits. For example, the ASCII codes for 'A', 'a', and '?' are 1000001, 1100001, and 0111111, respectively[1]. ASCII has long been superseded by the much more modern and extensive UTF-8 encoding method; ASCII codes constitute a very small subset of UTF-8 codes.

A *password* can only contain printable characters that appear on a standard US English keyboard. These include 26 upper case letters, 26 lower case letters, 10 numerical digits and the 32 special characters `‘’".,;:!?~@#$%^&*_-+=(){}[]<>\/|`. Not all ASCII codes correspond to printable characters.

a. (2 marks) What is the total number of ASCII encodings of 8-character strings?

b. What is the number of passwords of length 8 consisting of
   (i) (2 marks) any printable characters?
   (ii) (2 marks) lower case letters only?

c. Approximately what percentage of 8-character ASCII encodings are passwords consisting of
   (i) (2 marks) any printable characters?
   (ii) (2 marks) lower case letters only?

d. Assuming that each character in a password is chosen equally likely, what is the entropy of the space of passwords consisting of
   (i) (2 marks) 8 printable characters?
   (ii) (2 marks) 8 lower case letters?

e. Suppose we want a password space with entropy 128, i.e. a total of 2^{128} passwords assuming that all passwords are equally likely.[2] What is the minimum password length, i.e. the minimum number of characters in a password, to achieve this entropy, assuming that passwords consist of
   (i) (3 marks) any printable characters?
   (ii) (3 marks) lower case letters only?

[1] Source: `https://www.ascii-code.com`.

[2] For modern cryptosystems, 2^{128} is a typical key space size. Of course the assumption that all passwords are equally likely is a stretch: most of us use only a small handful of our favourite passwords.

Problem 2 — One-time pad without the all-zeros key (6 marks)

Recall that in the one-time pad, plaintexts, ciphertexts and keys are n-bit blocks for some $n \in \mathbb{N}$. For any key K, plaintext block M and ciphertext block C, the encryption of M with key K is $E_K(M) = M \oplus K$ and decryption of C with key K is $D_K(C) = C \oplus K$, where $\oplus$ denotes bitwise x-or (or equivalently, bitwise addition modulo 2). Assume as always that keys are chosen with equal likelihood.

When using the one-time pad with the key $K = 0^n$ consisting of n zeros, we have $E_K(M) = M \oplus K = M$ for every paintext M, i.e. M is left unencrypted. It has been suggested to improve the security of the one-time pad by only encrypting with non-zero keys, i.e. removing 0^n from the key space.

a. (4 marks) Use either the definition of perfect secrecy,

$$p(M) = p(M|C) \text{ for all plaintexts } M \text{ and ciphertexts } C \text{ with } p(C) > 0$$

or its equivalent characterization

$$p(C) = p(C|M) \text{ for all plaintexts } M \text{ with } p(M) > 0 \text{ and ciphertexts } C \text{ with } p(C) > 0$$

to *formally prove* that this modification of the one-time pad does not provide perfect secrecy. Specifically, you will need to give a counterexample, i.e. exhibit a pair (M, C) that violates the definition or the above characterization of perfect secrecy. Answers that argue informally or do not use this characterization will receive very little if any credit.

b. (2 marks) Explain (informally) why allowing $K = 0^n$ does not weaken the security of the one-time pad, even though using that key does not hide the plaintext when encrypting.

(*Note:* the assertion that choosing $K = 0^n$ happens so rarely that we don't have to worry about it is *not* a valid answer and will not receive any credit.)

Problem 3 — Weak collisions (16 marks)

The cryptographic relevance of this problem will become evident when we cover hash functions in class.

For each question below, provide a brief explanation and a compact formula for your answer. For part (d), give a numerical value that is an integer.

Let n be positive integer. Consider an experiment involving a group of participants, where we assign each participant a number that is randomly chosen from the set $\{1, 2, \ldots, n\}$ (so all these assignments are independent events). Note that we allow for the possibility of assigning the same number to two different participants.

Now pick your favourite number N between 1 and n. When any one of the participants is assigned the number N, we refer to this as a *weak collision* (with N). In this problem, we determine how to ensure at least a 50% chance of a weak collision in our experiment.

a. (2 marks) What is the probability that a given participant is assigned your favourite number N?

b. (2 marks) What is the probability that a given participant is not assigned the number N?

c. (3 marks) Suppose k people participate in the experiment (for some positive integer k). What is the probability that none of them is assigned the number N, i.e. that there is no weak collision?

d. (4 marks) Intuitively, the more people participate, the likelier we encounter a weak collision. We wish to find the minimum number K of participants required to ensure at least a 50% chance of a weak collision.

   Suppose $n = 10$. What is the threshold K in this case?

e. (5 marks) Generalizing part (d) from $n = 10$ to arbitrary n, prove that if the number of participants is above $\log(2)n \approx 0.69n$, then there is at least a 50% chance of a weak collision. Use (without proof) the inequality

$$1 - x < \exp(-x) \quad \text{for } x > 0 \; . \tag{1}$$

   (This inequality comes from the Taylor series

$$\exp(-x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \cdots + \cdots \; ,$$

   and the sum of the terms from $x^2/2$ onwards is positive.)

Concluding Remark. In (1), when x is small, $\exp(-x)$ is very close to $1 - x$; for example, $\exp(0.01)$ and $1 - 0.01$ are within 4 decimal places of each other. This implies that for large n, the threshold $0.69n$ of part (e) is close to optimal, i.e. we expect the necessary and sufficient number of participants for ensuring a weak collision with at least 50% probability to be on the order of n. Compare this to the corresponding result for (strong) collisions derived in the next problem.

Problem 4 — (Strong) collisions (20 marks)

The results of this problem bear relevance to Problem 7 below. Further cryptographic significance will become evident when we cover hash functions in class.

For each question below, provide a brief explanation and a compact formula for your answer. For part (c), give a numerical value that is an integer.

We consider the same experiment as in the previous problem, although here, you don't need to pick a favourite number N. When at least two of the participants are assigned identical numbers, we refer to this as a *strong collision* or just a *collision*. In this problem, we determine how to ensure at least a 50% chance of a collision in our experiment.

a. (4 marks) What is the probability that among k participants, no collision occurs?

b. (2 marks) What is the probability of a collision among k participants?

c. (4 marks) Once again, intuitively, the more people participate, the likelier we encounter a collision. We wish to find the minimum number K of participants required to ensure at least a 50% chance of a collision.

   Suppose $n = 10$. What is the the threshold K in this case?

d. (5 marks) Let P be the probability of no collisions as computed in part (a). Prove that

$$P \leq \exp\left(-\frac{k(k-1}{2n}\right) \ .$$

   First, if your expression obtained for P in part (a) is not already in this form, rewrite P as

$$P = \prod_{i=1}^{k-1}(1 - z_i) \ ,$$

   where $0 < z_i < 1$ for $1 \leq i \leq k-1$ (you'll have to figure out the appropriate expression for z_i). Then use inequality (1) from the previous problem.

e. (5 marks) Similarly to the previous problem, when n is much larger than the number k of participants, the upper bound on P in part (d) is a very close approximation to P. When in addition k is not too small (but still much smaller than n), k is very close to $k-1$. This means that the quantity $\exp(-k^2/2n)$ is a very close approximation to the probability P of part (a).

   Generalizing part (c) from $n = 10$ to arbitrary n, prove that if the number of participants is above $\sqrt{\log(4)n} \approx 1.177\sqrt{n}$, then there is at least a 50% chance of a collision. You may replace the actual expression for P derived in part (a) by $\exp(-k^2/2n)$.

   Note: you can solve this question even if you didn't attempt parts (a)-(d).

Concluding Remark. As in the previous problem, for large n, the threshold $1.177\sqrt{n}$ of part (e) is close to optimal, i.e. we expect the necessary and sufficient number of participants for ensuring a collision with at least 50% probability to be on the order of $\sqrt{n}$. Compare this to the corresponding result for weak collisions derived in the previous problem.

Written Problems for MATH 318 only

Submit your answers to this problem in a separate PDF file. **Do *not* include the solution to this problem in the PDF file containing your solutions to Problems 1-4.**

Let $\mathbb{F}_2 = \{0, 1\}$ with the usual arithmetic modulo 2. The set $\mathbb{F}_2^n = \{0, 1\}^n$ consisting of all n-bit vectors (vectors of length n with entries 0 or 1) is an n-dimensional vector space over $\mathbb{F}_2$, with the standard basis for example. Linear algebra in $\mathbb{F}_2^n$ as a vector space over $\mathbb{F}_2$ works just like linear algebra in $\mathbb{R}^n$ as a vector space over $\mathbb{R}$, except that linear combinations of vectors in $\mathbb{F}_2^n$ have coefficients 0 and 1, and all arithmetic is done modulo 2.

For any $n \in \mathbb{N}$, let $\mathrm{Gl}_n(\mathbb{F}_2)$ denote the set of invertible $n \times n$ matrices with zeros and ones as entries. Calculations involving such matrices again work exactly the same as the familiar matrix arithmetic over $\mathbb{R}$ that you learned in first year linear algebra, except that arithmetic using real numbers is again replaced by arithmetic modulo 2.

Now fix $n \in \mathbb{N}$ and consider the class of linear ciphers with $\mathcal{M} = \mathcal{C} = \mathbb{F}_2^n$, $\mathcal{K} = \mathrm{Gl}_n(\mathbb{F}_2)$, and for all plaintexts $\vec{m}$ (interpreted as n-bit column vectors with entries 0 and 1), encryption under a key matrix K is

$$E_K(\vec{m}) = K\vec{m} \ . \tag{2}$$

Then for all ciphertexts $\vec{c}$, decryption under K is obviously

$$D_K(\vec{c}) = K^{-1}\vec{c} \, .$$

Problem 5 — Transposition ciphers are linear (8 marks)

Recall that a *transposition cipher* produces a ciphertext by permuting the plaintext symbols. For a transposition cipher operating on bit strings of some length $n \in \mathbb{N}$, encryption of a bit string $(m_1, m_2, \ldots, m_n)$ can thus be written as

$$E_\sigma(m_1, m_2, \ldots, m_n) = (m_{\sigma(1)}, m_{\sigma(2)}, \ldots, m_{\sigma(n)}) \ ,$$

where σ is a secret key permutation on n symbols.

a. (3 marks) Let $n = 4$ and let σ be the permutation sending 1 to 2, 2 to 4, 3 to 3 and 4 to 1. Then transposition encryption of a 4-bit string using the key permutation σ is given by

$$E_\sigma(m_1, m_2, m_3, m_4) = (m_2, m_4, m_1, m_3) \ . \tag{3}$$

Now write 4-bit strings (x_1, x_2, x_3, x_4) as column vectors $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$. Prove that the cipher given in (3) is a linear cipher by explicitly writing down a *permutation* matrix K that satisfies (2) for this cipher. A permutation matrix is a a square matrix whose entries consist of zeros and ones such that there is exactly one one and $n - 1$ zeros in each row and in each column.

b. (5 marks) Generalizing part (a), formally prove that a transposition cipher operating on bit strings of any length n is a special case of a linear cipher where the matrix K in (2) is a permutation matrix. Use formal reasoning and good notation for your proof.

Problem 6 — Cryptanalysis of linear ciphers (30 marks)

In this problem, we explore attacks on linear ciphers as given in (2).

a. (3 marks) If each key matrix K is chosen equally likely, does this class of linear ciphers provide perfect secrecy? Formally prove your claim.

b. (3 marks) Explain how an attacker Eve can mount a chosen plaintext attack on a cipher of the form (2). The goal of this attack is to chose one or more plaintexts, obtain their encryptions under some unknown key matrix K, and derive K. How should Eve choose her plaintexts, and how many does she need to choose in order to be successful?

c. (3 marks) Let $\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_i$ be any collection of i linearly independent vectors in $\mathbb{F}_2^n$ for some i with $1 \leq i \leq n$. Prove that there are 2^i vectors $\vec{m}_{i+1} \in \mathbb{F}_2^n$ such that the vectors $\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_i, \vec{m}_{i+1}$ are linearly dependent.

d. (2 marks) As in part (c), let $\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_i$ be linearly independent vectors in $\mathbb{F}_2^n$. How many vectors $\vec{m}_{i+1} \in \mathbb{F}_2^n$ are there so that the vectors $\vec{m}_1, \vec{m}_2, \ldots, \vec{m}_i, \vec{m}_{i+1}$ are linearly independent?

e. (5 marks) Prove that the number of sets consisting of n linearly independent vectors in $\mathbb{F}_2^n$ is

$$\frac{1}{n!} \prod_{i=0}^{n-1} (2^n - 2^i) \ .$$

f. (5 marks) Prove that the probability that any set of n vectors in $\mathbb{F}_2^n$ is linearly independent is

$$P_n = \frac{\prod_{i=0}^{n-1} (2^n - 2^i)}{\prod_{i=0}^{n-1} (2^n - i)} \ .$$

g. (2 marks) Suppose $n = 4$. What is the probability that any set of 4 vectors in $\mathbb{F}_2^4$ is linearly *dependent*? Express the result as a fraction.

h. (6 marks) Part (b) considered a *chosen* plaintext attack, whereas we will now consider the scenario of mounting a *known* plaintext attack on a linear cipher as given in (2). Given a set of known plaintext/ciphertext pairs where all the plaintexts were encrypted to their respective corresponding ciphertexts using the same unknown key matrix K, the goal of this attack is to find K.

Assume that linear dependence of any collection of n plaintexts in $\mathbb{F}_2^n$ represents independent events, i.e. if p is the probability that any n plaintexts are linearly dependent, then p^2 is the probability that any two collections of n plaintexts are linearly dependent. Explain how Eve can use multiple attempts at a known plaintext attack to find the key matrix. For $n = 4$, how often does Eve have to try to achieve a 99 percent chance of success? In other words, what is the maximum number k of failed attempts such that Eve has a 99% chance of finding K on the $(k+1)$-st attempt?

Programming Problems for CPSC 418 only

Specifications. Design and implement your solutions using `Python 3` and the template programs provided on the Piazza "Resources" tab. **Please do not to alter the filenames, functions, or their input and output types.** Submissions that do not comply with these specifications will be penalized or not marked at all. Use the latest version of the `Python cryptography` library found at

`https://cryptography.io/en/latest/`

This link can also be found on the "references" page of our course website. Use this library for *all required cryptographic primitives*. This includes encryption, decryption, hashing and the PKCS7 `padding` module.

The `cryptography` library has its own interface to which you are expected to adhere. You must make use of the *hazardous materials layer*, **not** the *recipes layer*. Make sure to use good coding practices. You can split your program up into multiple source files, so long as the main executable retains the same name as the template program.

The testing of your program will largely be done on Gradescope using `python3`. An autograder will be made available to test component functions, while the overall problem solutions will be tested by the TAs separately.

You may assume that for any input file requirements, Gradescope will use text files of at most 1 MB in size and that all byte encoding is done using UTF-8.

Submission. Submit a description of your implementation for both problems in a separate README file in text format. **Do *not* include any written portions of the programming problems in the PDF file containing your solutions to any of the written problems.** Your description must include the following:

- A list of the files you have submitted that pertain to the problem, and a short description of each file.
- A list of what is implemented in the event that you are submitting a partial solution, or a statement that the problem is solved in full.
- A list of what is not implemented in the event that you are submitting a partial solution.
- A list of known bugs, or a statement that there are no known bugs.
- Any other answers to questions specified in the problem.

Problem 7 — Prefix collision finder (12 marks)

Overview. Bob is interested in storing the passwords for users of his website in hashed form using SHA-2 224 to produce the hashed outputs. However, for space reasons, he is considering storing only the first 5 bytes of each SHA-2 224 hash tag. As his friend, you want to convince Bob that this is a terrible idea by demonstrating that you can easily find two inputs whose SHA2 224 outputs have the same first 5 bytes.

Problem. Finish the template program named

`prefix_collision.py`

which finds two distinct **bytes** objects whose images under SHA-2 224 have the same 5-byte (40-bit) prefix.

Specifications. You will need to complete three separate functions:

1. `hash_bytes()`, which takes a sequence of bytes and returns a SHA-2 224 hash value.
2. `compare_bytes()`, which compares the first n bytes of two sequences and checks if they match.
3. `find_collision()`, which uses the other functions to find two byte sequences with hash values that match for the first n bytes.[3]

In order for your code to run efficiently you must utilize memory in a smart way. The number of SHA-2 224 tags that you compute and store should be on the order of $\sqrt{b}$, where b is the number if *bits* in the prefix. You should also make use of an appropriate data structure for quickly detecting duplicates.

Submission. Please submit a completed version of the template program, with the filename

`prefix_collision.py`

If you've spread your code across multiple source files, submit all of them. If your code takes more than 2 minutes on Gradescope's servers for a 5-byte prefix ($n = 5$), it will be timed out, so you will need to do something smarter than hashing every single input one-by-one.

Problem 8 — Password attack on authenticated encryption (26 marks)

Overview. Suppose Bob has designed his own authenticated encryption scheme, using the hash-then-encrypt paradigm, with AES-128 in CBC mode for encryption, and SHA-2 224 for the key derivation function and message hash tag. On input the name of a plaintext file and a password p, Bob's program does the following:

1. Converts the plaintext file to a byte array B.

[3] If you're an experienced coder, you might be happy to learn that we won't verify you actually used those prior functions to help create this one. They exist to provide guidance and partial marks for beginning coders.

2. Computes a hash tag t on the plaintext by applying SHA-2 224 to the byte array B, then appends t to B to obtain an extended byte array $B' = B||t$ (here, as always, "||" denotes concatenation).
3. Derives an encryption key by applying SHA-2 224 to the password p and truncating the result to the appropriate length for use in AES-128.
4. Generates a random 16-byte initial value IV (for use in CBC mode) and writes it to a file F.
5. Pads the extended byte array B' using the PKCS7 format if necessary, then encrypts the padded array with AES-128-CBC and appends the resulting ciphertext to the file F.

Out of laziness, Bob is known to use *strings* of the form YYYYMMDD, which mark certain dates from his life, as his passwords. Bob was born in the year 1984, and always includes the string FOXHOUND in his communications.

Problem. Create a Python 3 program that takes as input a file produced by Bob's hash-then-encrypt routine and performs the following tasks:

1. Determines the password used to derive the encryption key, and prints it out.
2. Decrypts the ciphertext using this key.
3. Checks the resulting plaintext for the phrase CODE-RED. If present, the program replaces this phrase with the phrase CODE-BLUE and writes the modified plaintext to a new file. If not present, the plaintext is left unchanged.
4. Computes a new hash tag on the possibly-modified plaintext.
5. Generate a new 16-byte IV.
6. Re-encrypt the possibly-modified plaintext plus new tag using the same password and exactly the same process as Bob's hash-then-encrypt program.
7. Writes the result to a new file. Recall that AES-128-CBC has a block size of 16 bytes.

Specifications. Fill in the empty functions in the template program `encrypt_modify.py`. Once complete, you should be able to replicate Bob's encryption scheme by running

```
python3 encrypt_modify.py --encrypt [PLAINTEXT] --password "YYYYMMDD"
                          [CYPHERTEXT]
```

where `PLAINTEXT` is the name of the file to by encrypted, `YYYYMMDD` is a number Bob would use for a password, and `CYPHERTEXT` is name of the file to store the output. Cracking the password, doing the text substitution, and reencrypting the result can be performed by running

```
python3 encrypt_modify.py --modify [ORIGINAL CYPHERTEXT] [NEW CYPHERTEXT]
```

where the input file `ORIGINAL CYPHERTEXT` is the output of the encryption process, and `NEW CYPHERTEXT` is an encrypted version of the possibly-modified plaintext. Don't forget to print out the password to the terminal.

In detail, the functions you'll need to fill in are:

1. `create_iv( length )`, which creates an IV that's `length` bytes long.
2. `derive_key( input )`, which takes a string and hashes it to form a key for AES-128.
3. `pad_bytes( input )`, which pads `input` so it can be encoded by AES-128-CBC.
4. `encrypt_bytes( input, key, iv )`, which encrypts `input` with the given key and IV.
5. `hash_pad_then_encrypt( input, string, iv )`, which uses the above functions to perform Bob's encryption algorithm.
6. `check_tag( input )`, which verifies the tag appended to `input` matches the hash of the remaining bytes.
7. `decrypt_unpad_check( input, string )`, which reverses Bob's encryption algorithm, and both verifies and strips off the tag.
8. `generate_passwords( year, month, day )`, which generates a list of passwords to try, according to Bob's format, with the earliest starting on `year/month/day`.
9. `determine_password( input )`, which tries to brute-force decrypt `input` using `generate_passwords( year, month, day )` and `decrypt_unpad_check( input, string )`.
10. `attempt_substitute( input, codeword, target, substitute )`, which combines all of the above to do the decryption, substitution, and re-encryption.

Additional details are in the template program `encrypt_modify.py`, on the Piazza resources page.

Submission. Please submit a completed version of the template program, with the filename

`encrypt_modify.py`

If you've spread your code across multiple source files, submit all of them.