CPSC 418 / MATH 318 Introduction to Cryptography

ASSIGNMENT 1

Due: **Thursday, October 7, 2021** at **11:55 PM** Total marks: **100+4 bonus marks**

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.

Problem 5 is worth 38 marks + 4 bonus marks and is for MATH 318 students only; CPSC 418 students may do this problem for limited extra credit.

Problem 6 is worth 38 marks + 4 bonus marks and is for CPSC 418 students only; MATH 318 students may do this problem for limited credit.

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

Problem 5 uses the letters $\mathcal{M}$, $\mathcal{C}$, $\mathcal{K}$, $\mathcal{O}$, $\mathcal{R}$ and $\mathcal{T}$ in caligraphy font. You can generate that font with the LaTeX command

$$|\mathcal{X} —$$

which produces the output $\mathcal{X}$. To avoid having to type such a rather lengthy command repeatedly, the Latex template provided includes aliases for these six letters. Specifically, `|$\M$|`produces $\mathcal{M}$, `|$\C$|`produces $\mathcal{C}$ etc.

Written Problems for CPSC 418 and MATH 318

Problem 1 — Linear feedback shift register key streams (11 marks)

Stream ciphers such as the one-time pad require a secret key stream of pseudorandom bits.In this problem, you will cryptanalyze one possible approach for generating such a key stream.

Let m be a positive integer and $c_0, c_1, \ldots, c_{m-1} \in \{0,1\}$ a sequence of m fixed bits. Let $z_0, z_1, \ldots, z_{m-1}$ be any sequence of m bits and define $z_m, z_{m+1}, z_{m+1}, \ldots$ via the linear recurrence

$$z_{n+m} \equiv c_{m-1}z_{n+m-1} + c_{m-2}z_{n+m-2} + \cdots + c_1 z_{n+1} + c_0 z_n \pmod 2 \ , \tag{1}$$

with the usual arithmetic modulo 2. The fixed bits $c_0, c_1, \ldots c_{m-1}$ are the *coefficients* of the linear recurrence (1) and the initial values $z_0, z_1, \ldots, z_{m-1}$ are its *seed*. If the seed and the coefficients are appropriately chosen, then (1) generates a sequence of 2^m pseudorandom bits[1] $(z_i)_{i \geq 0}$ from a seed of length m. This type of construction is popular since it can be implemented very efficiently in hardware using a *linear feedback shift register*; see pp. 36-37 of the Stinson-Paterson book.

a. (2 marks) Consider the recurrence

$$z_{n+5} \equiv z_{n+2} + z_n \pmod 2 \ ,$$

This represents the special case of (1) with $m = 5$ and $(c_4, c_3, c_3, c_1, c_0) = (0, 0, 1, 0, 1)$. Write down the first 31 bits $z_0, z_1, \ldots, z_{30}$ generated by this recurrence with seed $(z_0, z_1, z_2, z_3, z_4) = (0, 0, 1, 0, 1)$.

b. (6 marks) Suppose you are given the 10-bit sequence

$$(z_3, z_4, z_5, z_6, z_7, z_8, z_9, z_{10}, z_{11}, z_{12}) = (1, 0, 0, 1, 1, 0, 1, 0, 0, 1)$$

which was generated using an unknown linear recurrence of the form (1) with $m = 5$, i.e.

$$z_{n+5} \equiv c_4 z_{n+4} + c_3 z_{n+3} + c_2 z_{n+2} + c_1 z_{n+1} + c_0 z_n \pmod 2 \ .$$

Find the unknown coefficients c_0, c_1, c_2, c_3, c_4 of this recurrence. Your answer *must use a systematic approach* and should make no assumptions or distinguish cases; e.g. solutions that include a phrase such as "suppose $c_3 = 0$" and then either derive a contradiction or obtain the values of c_0, c_1, c_2, c_4 will get zero credit. Show your work and remember your linear algebra!

Suggestion: check your answer, i.e. ensure that the coefficients you obtain define a recurrence that produces the second five given bits $(z_8, z_9, z_{10}, z_{11}, z_{12}) = (0, 1, 0, 0, 1)$ from the first four given bits $(z_3, z_4, z_5, z_6, z_7) = (1, 0, 0, 1, 1)$.

Note: the hardest part of this problem is probably typesetting it properly; hence the high mark score. So this is a good problem on which to hone your LaTeX skills. Use math display environments like `align*` to nicely line up your congruences and learn how to typeset matrices.

c. (3 marks) Suppose you are intercepting a bit stream which you know as generated via a recurrence relation of the form (1) with $m = 5$. Through the method illustrated in parts (b) and (c), you were able to find out that the recurrence is

$$z_{n+5} \equiv z_{n+4} + z_{n+3} \pmod 2 \qquad (n \geq 0) \ . \tag{2}$$

[1]Since there are only $2^m - 1$ distinct non-zero bit patterns of length m, there must be repetition after at most $2^m - 1$ bits. So in practice, m must be large.

Now suppose this bit stream is used to generate keys for encryption with the one-time pad. You intercept the 10-bit ciphertext fragment $C = 1001101000$ that was one-time pad encrypted using as key a 10-bit sequence $(z_0, z_1, \ldots, z_9)$, where $(z_0, z_1, z_2, z_3, z_4)$ is an unknown seed and the 5-tuple $(z_5, z_6, z_7, z_8, z_9)$ was obtained from this seed using the recurrence (2). Suppose also that you discover that the last two bits of the seed (i.e. z_3 and z_4) are 1 and 0, respectively. Decrypt as many bits of C as possible.

Problem 2 — Password counts (15 marks)

In each question below, provide a brief explanation, a formula and the numerical value for your answer (exact if the answer is an integer, approximate otherwise).

There are 94 *printable characters* on a standard North American keyboard, comprised of the 26 upper case letters A-Z, the 26 lower case letters a-z, the 10 numerical digits 0-9 and the 32 special characters

```
‘ ’ " . , ; : ! ? ~ @ # $ % ^ & * _ - + = ( ) { } [ ] < > \ / |
```

For our purposes, *passwords* are strings consisting of printable characters. The *length* of a password is the number of characters in the password.

a. (1 mark) What is the total number of passwords of length 8?

b. (3 marks) Suppose a user has a password of length 8 whose first four characters are the first four letters of their oldest child's name (all in either lower or upper case) and the second four characters are the child's birthday in the format[2] DDMM. Intelligence gathering on your part reveals that the child's first name starts with 'L' and that the child was born in 2008. Making no further assumption about a "reasonable" first name (e.g., for all you know, the kid's name could be 'Lxqscdx'), what is the minimal number of candidates that our lazy user could potentially be using as their password?

c. (4 marks) To guard against *dictionary attacks* (where a password cracker checks if a password is a common word or phrase), passwords are typically required to contain at least one numerical digit and at least one special character. What is the total number of passwords of length 8 that satisfy this requirement?

(*Hint:* It is easier to characterize the number of passwords that violate this rule and subtract that count from the total number of 8-character passwords. But be careful that you don't subtract out some passwords twice.)

d. (2 marks) What is the percentage of 8-character passwords that satisfy the rule of part (c)?

e. (2 marks) Assuming that each permissable character in a password is chosen equally likely[3], what is the entropy of the password space of

[2]For example, if a user's oldest child's name was Robin and Robin's birthday was on July 1, then the user's password would be either ROBI0107 or robi0107, but not Robi0107 or rObI0107.

[3]This assumption may be appropriate for passwords chosen by computer systems with good random number generators, but it is utterly false for passwords chosen by humans. So in practice, the minimum password length computed in part (f) is a significant underestimate.

- part (a)?
- part (c)?

f. (3 marks) Suppose we want a password space with entropy 128 bits, assuming that keys are chosen equally likely, i.e. a total of 2^{128} passwords (this number is typical for key space sizes of modern cryptosystems). Assuming no restrictions on the characters appearing in passwords (i.e. the scenario of part (a)), what is the minimum password length that guarantees a password space with entropy 128?

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.

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? Give a numerical value that is an integer.

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[4]

$$1 - x < \exp(-x) \quad \text{for } x > 0 \; . \tag{3}$$

Concluding Remark. In (3), 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.

[4]This inequality comes from the Taylor series $\exp(-x) = 1 - x + \dfrac{x^2}{2!} - \dfrac{x^3}{3!} + \dfrac{x^4}{4!} - \cdots + \cdots$, and the sum of the terms from $x^2/2$ onwards is positive.

Problem 4 — (Strong) collisions (20 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.

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? Give a numerical value that is an integer.

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 (3) 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 threshold of $0.69n$ (i.e. of order n) for weak collisions derived in the previous problem. Strong collisions are expected to be much more likely. E.g. drawing just 12 numbers between 1 and 100 is more likely than not to produce a strong collision, whereas it takes as many as 69 draws to make a weak collision more likely than not.

Written Problem 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.**

Problem 5 — A time-memory trade-off for key search (38 marks + 4 bonus marks)

This problem investigates a time-memory trade-off for exhaustive key search of a cipher. Normally, searching a key space of size n takes up to n test decryptions but requires no memory as every wrong decryption (not yielding the valid plaintext) can immediately be discarded. The strategy in this problem requires only t such tests, at the expense of storing on the order of n/t keys (though frequently fewer keys than that). A drawback is that the search strategy requires a very expensive pre-computation of tn tests. However, this pre-computation need only be done once and its results can be re-used for an arbitrary number of subsequent key searches. Here, t is a *trade-off parameter*, which is a number between 2 and n chosen by the attacker. The choice of t needs to balance the feasibility of the pre-computation, the amount of available memory, and the amount of time to be spent on key search.

We assume that the message space, ciphertext space and key space for this cryptosystem are all identical, i.e. $\mathcal{M} = \mathcal{C} = \mathcal{K}$. For brevity, put $n = |\mathcal{K}|$. We also assume that for every plaintext/ciphertext pair (M, C), there is a unique key K such that $E_K(M) = C$. Examples of ciphers that satisfy these properties include the shift cipher, the one-time pad, the Advanced Encryption Standard (AES) and many others.

Our attack scenario is a known plaintext attack, where the adversary has a plaintext/ciphertext pair (M, C) and the task is to search for the unique unknown key K such that $E_K(M) = C$. We define a function

$$g : \mathcal{K} \to \mathcal{K} \quad \text{via} \quad g(K) = E_K(M) \ ,$$

where we note that the co-domain of g is $\mathcal{C} = \mathcal{K}$. So we are searching for the key $K \in \mathcal{K}$ such that $g(K) = C$.

In order to understand and illustrate the attack, we will use as an example a cryptosystem that represents a modification of the shift cipher on an alphabet of $n = 51$ characters. Specifically, $\mathcal{M} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_{51} = \{0, 1, \ldots, 50\}$ (the integers modulo 51) and

$$E_K(M) \equiv M + 8K \pmod{51} \ , \qquad D_K(C) \equiv C - 8K \equiv C + 43K \pmod{51} \tag{4}$$

for all $M, C, K \in \mathbb{Z}_{51}$.

Some parts of this problem ask you to compute quantities specific to the this sample cipher. All parts not stating explicitly that you should use the cipher defined in (4) should be solved in general, i.e. for any cipher as described above. *Even if you are having difficulties answering the general questions, you are urged to attempt the parts that refer specifically to the cipher of (4).*

a. (2 marks) Prove that the map g is a bijection on $\mathcal{K}$.

b. As usual, for $K \in \mathcal{K}$ and any $i \in \mathbb{Z}$, let $g^i(K)$ denote

- g applied i times to K when $i > 0$,
- K when $i = 0$ (i.e. g^0 is the identity on $\mathcal{K}$),
- the inverse g^{-1} applied $|i|$ times to K when $i < 0$.

For any $K \in \mathcal{K}$, the *orbit* of K under g is the subset

$$\mathcal{O} = \{g^i(K) \mid i \in \mathbb{Z}\} .$$

Since $\mathcal{K}$ is finite, orbits are actually finite sets containing at most n elements.

(i) (5 marks) Prove that for every orbit $\mathcal{O}$, there exists a positive integer $n_{\mathcal{O}}$ such that $\mathcal{O}$ can be written in the form

$$\mathcal{O} = \{K, g(K), g^2(K), \dots, g^{n_{\mathcal{O}}-1}(K)\} = \{g^k(K) \mid 0 \le k \le n_{\mathcal{O}} - 1\} , \tag{5}$$

where K is any key in $\mathcal{O}$ and the elements $g^i(K)$ with $0 \le k \le n_{\mathcal{O}} - 1$ are all distinct. The number $n_{\mathcal{O}}$ of elements in $\mathcal{O}$ is called the *length* of $\mathcal{O}$.

(ii) (3 marks) Prove that $\mathcal{K}$ is the union of disjoint orbits.

(iii) (3 marks) Compute the orbits for the cipher given in (4) using $M = 1$. For every orbit $\mathcal{O}$, start with the smallest element $K \in \mathbb{Z}_{51}$ and write down the distinct elements in $\mathcal{O}$ in the order $K, g(K), g^2(K), \dots$

c. Let t be the trade-off parameter. Call an orbit $\mathcal{O}$ *short* if its length is at most t (i.e. $n_{\mathcal{O}} \le t$) and *long* otherwise (i.e. $n_{\mathcal{O}} \ge t + 1$).

(i) (1 mark) For $t = 3$, identify the short and long orbits for the cipher given in (4).

(ii) (2 marks) Prove that the number of long orbits[5] is at most $n/(t + 1)$.

d. For any key $K \in \mathcal{K}$, define the set

$$\mathcal{R}(K) = \{g^t(K), g^{2t}(K), \dots, g^{q_{\mathcal{O}}t}(K)\} = \{g^{it}(K) \mid 1 \le i \le q_{\mathcal{O}}\} ,$$

where $n_{\mathcal{O}}$ is the length of the orbit $\mathcal{O}$ containing K and $q_{\mathcal{O}}$ is the unique integer such that $(q_{\mathcal{O}} - 1)t < n_{\mathcal{O}} \le q_{\mathcal{O}}t$ (or equivalently, $(q_{\mathcal{O}} - 1)t + 1 \le n_{\mathcal{O}} \le q_0 t$ as $n_{\mathcal{O}}$ and $(q_{\mathcal{O}} - 1)t$ are integers).

(i) (1 mark) Prove that for every key $K \in \mathcal{K}$, $\mathcal{R}(K)$ has cardinality at most $(n_{\mathcal{O}} - 1)/t + 1$, where $n_{\mathcal{O}}$ is the length of $\mathcal{O}$.

(ii) (3 marks) For $t = 3$, compute the collection of sets $\mathcal{R}(K)$ of the cipher given in (4), Specifically, for each $\mathcal{O}$ computed in part (b) (iii), write down the elements of $\mathcal{R}(K)$ in the order $g^3(K), g^6(K), \dots$, where K is the smallest element of $\mathcal{O}$.

e. Let $\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_r$ be all the distinct *long* orbits. For each i with $1 \le i \le r$, fix a key $k_i \in \mathcal{O}_i$ and put

$$\mathcal{R} = \mathcal{R}(k_1) \cup \mathcal{R}(k_2) \cup \cdots \cup \mathcal{R}(k_r) .$$

[5] This bound can be very crude in practice if all the long orbits are much longer than $t + 1$. For our sample cipher (4) for example, we have $n/(t + 1) = 51/4 = 12.75$, but the number of long orbits is significantly less than 12.

(i) (2 marks) Compute the set $\mathcal{R}$ for the cipher given in (4) using the keys k_i where, as before, each k_i is the smallest element in its (long) orbit. List the elements of $\mathcal{R}$ in ascending order.

(ii) (3 marks) Let $K \in \mathcal{K}$ be a key contained in some long orbit. Prove that there exists $R \in \mathcal{R}$ with $R \neq K$ such that $R \in \{g(K), g^2(K), \ldots, g^t(K)\}$. Intuitively, this means that every key $K \in \mathcal{K}$ is either contained in an orbit of length at most t or is at most t applications of g away from from an element in $\mathcal{R}$.

(iii) (4 marks) Prove that $\mathcal{R}$ has cardinality[6] at most $2n/(t+1)$.

f. Define the set

$$\mathcal{T} = \{(K, g^{-t}(K)) \mid K \in \mathcal{R}\} .$$

(i) (2 marks) Compute the set $\mathcal{T}$ for $t = 3$ for the cipher defined in (4) using the set $\mathcal{R}$ computed in part (e) (i). List the elements of $\mathcal{T}$ in ascending order of first coordinate, i.e. in the same order as the elements in $\mathcal{R}$.

(ii) **Bonus** (4 marks) Describe a procedure that computes the sets $\mathcal{R}$ and $\mathcal{T}$ using no more than nt application of the function g and without having to store all n keys (or any other array of size n) in memory. Prove that your procedure computes $\mathcal{R}$ and $\mathcal{T}$ correctly and complies with the specified time and memory restrictions.

g. Consider the following search procedure that takes as input the pair (M, C) and the sets $\mathcal{R}, \mathcal{T}$ and finds the key K such that $C = E_K(M)$:

Starting with $K_0 = C$, compute the elements $K_1 = g(K_0)$, $K_2 = g(K_1)$, ... until one one of the following conditions is satisfied:

A) If an index $i > 0$ is found such that $K_i = C$, output $K = K_{i-1}$.

B) If an index $i \geq 0$ is found such that $K_i \in \mathcal{R}$, look up the entry with first coordinate K_i in $\mathcal{T}$ and put $X = g^{-t}(K_i)$ (the second coordinate of that entry in $\mathcal{T}$). Starting with $K'_0 = X$, compute the elements $K'_1 = g(K'_0)$, $K'_2 = g(K'_1)$, ... until a key K'_j is found such that $K'_j = C$. Output $K = K'_{j-1}$.

(i) (3 marks) Execute the search procedure for the cipher given in (4) with $t = 3$ using the ciphertexts $C = 12$, $C = 29$ and $C = 37$. For each C, indicate which of the conditions (A) or (B) is saisfied, and write down all the keys K_i computed during the search, as well as the output key K.

Suggestion: check your output keys against (4) to make sure that you have found the correct key that encrypts $M = 1$ to C.

(ii) (4 marks) Prove that the search procedure outputs the correct key and performs no more than t evaluations of the map g.

Note: the computational effort of comparing keys to C in condition (A), checking whether $C \in \mathcal{R}$ in condition (B) and looking up the entry in $\mathcal{T}$ that has first coordinate C as per condition (B) is generally negligible compared to the evaluations of g. These kinds of operations on finite sets can be implemented very efficiently via hash tables, for example. So we ignore that cost in our analysis.

[6] As pointed out in the previous footnote, much of the time the set $\mathcal{R}$ is much smaller because the bound $2n/(t+1)$ here uses a bound on the number of long orbits that is frequently a significant overestimate.

Programming Problem for CPSC 418 only

Problem 6 — Create Your Own Encryption Algorithm (38 marks + 4 bonus marks)

Overview

For many years after World War II, cryptographic algorithms were considered equivalent to missiles by the United States government. Their export has been tightly regulated, even when it resulted in absurdities like illegal T-shirts. While the current regulations are much more permissive than they have been in the past, a mere a flow chart of the current regulations is daunting.

To this day, products containing strong encryption cannot be sold by US citizens to anyone in five countries. But consider the plight of a dissident within such a country. This ban severely restricts their ability to protect themselves from their government via encryption. Their government, in contrast, has sufficient resources to evade the ban. While the export restrictions are well-meaning, they seem to give "rogue nations" an advantage over their citizens.

Our hypothetical dissident has an ace up their sleeve, however: they can design their own encryption algorithm by carefully combining multiple cryptographic primitives that will be introduced in class. Ironically, this dissident can use algorithms developed in the United States to create an arbitrarily strong encryption algorithm.

Problem Statement

Complete the template program named

`encrypt_decrypt.py`

which has the primary goal of encrypting or decrypting a file using your algorithm.

Specifications

Design and implement your solution using `Python 3` and the template program provided on the "Resources" tab of Piazza. **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.

The testing of your program will largely be done on Gradescope using `python3`. An autograder will be made available to test component functionality, and at the minimum a TA will review your code.

You may assume that any encoding of strings into byte representations and vice-versa is done using UTF-8.

You will need to complete nine separate functions and one variable:

1. `generate_iv( length )`, a function that returns a random number `length` bytes long that was derived from a cryptographically secure source.

2. `Hasher.hash( data )`, a function that takes in an arbitrarily long sequence of bytes and uses a cryptographically secure hash function to reduce it to a digest of `Hasher.digest_bytes` or 32 bytes in length.

3. `Hasher.block_bytes`, a variable that contains the ideal byte size of blocks to feed into `Hasher.hash()`.

4. `xor( first, second )`, a function that takes two arbitrarily-long sequences of bytes and exclusive-ors them together into a third sequence of bytes. `first` or `second` are padded with zeros if they differ in length.

5. `HMAC( data, key, Hasher )`, a function that implements the HMAC algorithm via `Hasher.hash()`, with the key `key` on the arbitrarily-long sequence of bytes `data`, returning the resulting digest.

6. `pad( data, digest_size )`, a function that performs the PKCS7 padding algorithm on the byte sequence `data`, such that its length is now a multiple of `digest_size`.

7. `unpad( data )`, a function that attempts to reverse the PKCS7 algorithm on `data`, either returning a byte sequence with all padding removed or `None` if the padding wasn't present or was invalid.

8. `encrypt( iv, data, key, Hasher, block_ids )`, a function that carries out the core encryption algorithm with IV *iv* on plaintext `data` using key `key`. It is helped by a `Hasher` instance and a generator `block_ids`.

9. `pad_encrypt_then_HMAC( plaintext, key_cypher, key_HMAC, hasher, block_ids )`, a function that extends `encrypt()` so that it can handle arbitrarily-long byte sequences, plus adds semantic security and message authentication.

10. `decrypt_and_verify( cyphertext, key_cypher, key_HMAC, hasher, block_ids )`, a function that reverses the process done by `pad_encrypt_then_HMAC()`, if possible. Integrity checking is mandatory.

Writing an encryption algorithm is challenging even for professionals, and it is generally recommended to use pre-existing implementations when possible. As a learning exercise, though, it ties together multiple tools covered in this class to form something greater than its parts.

To make things easier, for instance, the problem of creating an encryption algorithm has been converted to that of writing a cryptographically secure hash function. This is also highly non-trivial, so we've provide a major helping hand: your implementation of `Hasher.hash()` can be a simple wrapper around an existing hash function. There are a number of Python libraries that can perform cryptographically secure hashing, at least one of which is included in the core libraries. We recommend you use SHA-256 as the hash function; other algorithms that are considered insecure (eg. SHA1) or with smaller digest sizes (SHA-224) will result in deducted marks.[7]

`encrypt()` should carry out the following steps:

1. Create a generator from `block_ids`.

2. Divide `data` into appropriately-sized blocks.

[7]The code has been structured to allow for digest sizes greater than 32 bytes, with some trivial tweaking of the template, but this won't earn extra marks and increases the odds of your code breaking.

3. For each block:
    (a) Pull a block ID from the generator.
    (b) Exclusive-or that ID with the initialization vector.
    (c) Feed that as input into `HMAC()`.
    (d) Exclusive-or the output of `HMAC()` with the plaintext block to create a ciphertext block.
4. Join all the ciphertext blocks into one continuous byte sequence and return that sequence.

`pad_encrypt_then_HMAC()` is as the name suggests: pad the plaintext, then encrypt the padded plaintext with the encryption key to obtain the ciphertext. Next, prepend the initialization vector to the ciphertext and then `HMAC()` the combination using the HMAC key. Finally, append the HMAC output, so that the final result will be

`IV` ‖ `ciphertext` ‖ `HMAC-tag`.

The `decrypt_and_verify()` function may seem impossible to write without a `decrypt()` function, but the lecture notes on the one-time pad will illustrate how this can be done.

To assist you, the template itself is a fully-functional program once your code is in place. For example,

```
encrypt_decrypt.py --encrypt plaintext.txt --output cyphertext.bin --password
123456
```

will encrypt `plaintext.txt` with the password ”123456“, and put the contents in a file named `cyphertext.bin`. Whereas

```
encrypt_decrypt.py --decrypt cyphertext.bin --reference plaintext.txt
```

will decrypt the file `cyphertext.bin` with the default password, attempt to print the output to the terminal, and double-check the output against a reference file to see if there are any differences.[8] The template is also well documented, and the supported command-line switches can be printed by executing

```
encrypt_decrypt.py --help
```

This functionality is very handy for testing your code if you cannot access the auto-grader, and even if you can it often results in more rapid feedback.

[8]The default password is not ”123456”, so if you executed these commands exactly, you should get an error saying the cyphertext could not be decrypted.

Bonus (4 marks)

Some of you may be looking at the single line you wrote for `Hasher.hash()`, and feeling like the assignment is too easy. Writing a novel cryptographically secure hash function is incredibly hard, though, both for the effort required to show it is plausibly secure as well as the constrained design space. Implementing an existing one on your own, without using a library, encourages copy-paste code.

Consider this compromise: implement an existing cryptographically secure hash function, but add at least one non-trivial change to the algorithm. For instance, many such hash functions involve repeating a fixed algorithm for a fixed number of rounds. Increasing or decreasing the number of rounds is trivial; changing a constant used by the round algorithm is usually non-trivial. A few algorithms invoke ”nothing-up-my-sleeve“ numbers to initialize these constants, though, and changing one of those is trivial. When in doubt, consult with a TA.

We will not require you to properly prove the changed algorithm is cryptographically secure but we will require you to show your algorithm can pass a randomness test. Your hash function will be passed the numeric series $\{0, 1, 2, \dots\}$ as input, the output concatenated together, and fed into the `dieharder` random test suite.[9] If no tests fail, and you've properly documented your hash function, you will receive full bonus marks.

"Properly documented" means you've listed the original cryptographically secure hash function you used as a basis as well as all the changes you've made, both trivial and non-trivial.

The `--dump` option allows both you and the TA to perform the above test. Something like

```
$ python3 encrypt_decrypt.py --dump | dd of=/tmp/test.bin bs=1k count=5M status=progress
$ dieharder -a -g 201 -f /tmp/test.bin > report.txt
```

will be executed to perform the aforementioned tests.[10]

Submission

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

`encrypt_decrypt.py`

If you've spread your code across multiple source files, submit all of them. The auto-grader will time out if your code takes more than ten minutes to execute; the reference code requires a fraction of that time to finish, so if you receive a timeout it is likely due to an infinite loop in your code or a slow choice of algorithm, rather than a bug in the auto-grader.

[9] `dieharder` was chosen because it is both convenient to use and less stringent than its main competitors, `PractRand` and `TestU01`. It should also be stressed that passing randomness tests is insufficient; most of Melissa O'Neil's PCG family pass all such tests, yet Bouillaguet (2020) demonstrated 512 bytes of output was sufficient to recover the initial seed value.

[10] While `dieharder` requires something like 250GB of data to perform a full test, some experimentation suggests 5GB gives the same results. This is handy, as it's quite likely your python hash function will execute much slower than the C equivalent, but the input data size may grow if further experimentation demands it. Also, these commands are specific to Linux and Mac OS X, and are unlikely to work on Windows as-is.

Also submit a description of your implementation for this problem in a separate README file in text format. **Do *not* include the written portion of the programming problem in the PDF file containing your solutions to 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. This should include a brief overview of how you implemented the requested algorithms.
- Either 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, if you instead have a full solution.
- 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.

Hints and Tips

- Cryptographic hash functions have two sizes associated with them, the input block size and output digest size, and these are almost always different. Confusing the two will cause algorithms to fail.
- If you are struggling with how to call or use certain functions, remember that the template is a functional program. It may contain examples of how to do exactly that.
- The autograder doesn't work by calling `encrypt_decrypt.py` directly. Instead, it imports the functions you've designed and calls them. This explains why we stressed not to change function names earlier, but it has a useful side effect: you can test your code the same way the auto-grader does.

```
$ python3
Python 3.9.6 (default, Jul 16 2021, 00:00:00)

>>> from encrypt_decrypt import Hasher, HMAC
>>> h = Hasher()   # uses SHA-256
>>> HMAC( b"The quick brown fox jumps over the lazy dog", b'key', h ).hex()
'f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8'
```