# CPSC 418 / MATH 318 — Introduction to Cryptography

# ASSIGNMENT 2

Due: **Wednesday, Mar. 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 1-5 are worth 63 marks and are required for both CPSC 418 and MATH 318 students.

Problems 6-8 are worth 37 marks and are for MATH 318 students only; CPSC 418 may do these for limited extra credit.

Problem 9 is worth 37 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 — Arithmetic in the AES MixColumns operation (20 marks)

Recall that the MixColumns operation in AES performs arithmetic on 4-byte vectors using the polynomial $M(y) = y^4 + 1$. In this arithmetic, we have $M(y) = 0$, so $y^4 = 1$.

a. In this part of the problem, we consider multiplication of 4-byte vectors (viewed as polynomials of degree $\leq 3$ whose coefficients are bytes) by powers of $y$.

   (i) (2 marks) Formally prove that in this arithmetic, multiplication of any 4-byte vector by $y$ is a circular left shift of the vector by one byte.

   (ii) (4 marks) Generalizing part (i) to other powers of $y$, formally prove that multiplication of any 4-byte vector by $y^i$ $(0 \leq i \leq 3)$ is a circular left shift of the vector by $i$ bytes.

b. Next, we consider arithmetic involving the coefficients of the polynomial

$$c(y) = (03)y^3 + (01)y^2 + (01)y + (02) \ ,$$

   that appears in MixColumns, where the coefficients of $c(y)$ are bytes written in hexadecimal (i.e. base 16) notation. Arithmetic involving this polynomial requires the computation of products involving the bytes (01), (02) and (03) in the Rijndahl field GF($2^8$). Recall that in this field, arithmetic is done modulo $m(x) = x^8 + x^4 + x^3 + x + 1$.

   (i) (2 marks) Write the bytes (01), (02), (03) as their respective polynomial representatives $c_1(x)$, $c_2(x)$ and $c_3(x)$ in the Rijndahl field GF($2^8$).

   (ii) (4 marks) Let $b = (b_7\, b_6 \cdots b_1\, b_0)$ be any byte, and let $d = (02)b$ be the product of the bytes (02) and $b$ in the Rijndahl field GF($2^8$). Then $d$ is again a byte of the form $d = (d_7\, d_6 \cdots d_1\, d_0)$. Provide formulas for the bits $d_i$, $0 \leq i \leq 7$, in terms of the bits $b_i$.

   (iii) (3 marks) Provide analogous expressions as in part (b) (ii) for the byte product $e = (03)b$, where $b = (b_7\, b_6 \cdots b_1\, b_0)$ is any byte.

c. The MixColumns operation performs multiplication of 4-byte vectors by the polynomial $c(y)$ of part (b). In this part of the problem, you will evaluate such products symbolically.

   (i) (3 marks) Let $s(y) = s_3y^3 + s_2y^2 + s_1y + s_0$ be a polynomial whose coefficients are bytes. Symbolically compute the product $t(y) = s(y)c(y) \bmod y^4 + 1$. The result should be a polynomial of the form $t(y) = t_3y^3 + t_2y^2 + t_1y + t_0$ where $t_3, t_2, t_1, t_0$ are bytes. Provide formulas for the bytes $t_i$, $0 \leq i \leq 3$, in terms of the bytes $s_i$. The equations should consist of sums of byte products of the form $01s_i$, $02s_i$, $03s_i$, $0 \leq i \leq 3$. You need *not* compute these individual byte products as you did in part (b).

   (ii) (2 marks) Write your solution of part (c) (i) in matrix form; i.e. give a $4 \times 4$ matrix $C$ whose entries are bytes such that

$$\begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix} = C \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} .$$

   Note that this yields the matrix representation of MixColumns presented (without proof) in class.

## Problem 2 — Error propagation in block cipher modes (12 marks)

Error propagation is often an important consideration when choosing a mode of operation in practice. In this problem, you will analyze the error propagation properties of an arbitrary block cipher in various such modes; note that these properties are independent of the cipher used.

a. Suppose Alice wants to send a sequence of message blocks $M_0, M_1, M_2, \dots$ to Bob, encrypted to ciphertext blocks $C_0, C_1, C_2, \dots$ using some fixed key $K$. Assume that an error occurs during transmission of a particular block of ciphertext $C_i$. Justifying all your answers, explain which plaintext blocks are affected after Bob decrypts this (faulty) ciphertext block $C_i$ when the cipher is used in

   (i) (2 marks) ECB mode?

   (ii) (2 marks) CBC mode?

   (iii) (2 marks) OFB mode?

   (iv) (2 marks) CFB mode with one register?

   (v) (2 marks) CTR mode?

b. (2 marks) Suppose now that an error occurs in a particular plaintext block $M_i$ before Alice encrypts it and sends the corresponding ciphertext $C_i$ to Bob. Upon decryption of $C_i$, which plaintext blocks $M_j$ are affected when using the cipher in CBC mode?

## Problem 3 — Binary exponentiation (13 marks)

Recall the exponentiation algorithm given in class for evaluating $a^n \pmod{m}$ $(a \in \mathbb{Z},\ m, n \in \mathbb{N})$:

1. Compute the binary representation of $n$:

$$n = b_0 2^k + b_1 2^{k-1} + \cdots + b_{k-1} 2 + b_k \ ,$$

   with $b_0 = 1$, $b_i \in \{0, 1\}$ for $1 \le i \le k$, and $k = \lfloor \log_2 n \rfloor$.

2. Initialize $r_0 \equiv a \pmod{m}$.
3. For $0 \le i \le k-1$ compute $r_{i+1} = \begin{cases} r_i^2 \pmod{m} & \text{if } b_{i+1} = 0 \ , \\ r_i^2 a \pmod{m} & \text{if } b_{i+1} = 1 \ . \end{cases}$
4. Output $r_k$.

a. (3 marks) To warm up with a toy example, compute $17^{11} \pmod{77}$ using the procedure above; answers that don't use the binary exponentiation algorithm will receive no credit, even if they are correct. Show all your work, and write down all your intermediate quantities $b_i$ and $r_i$. Your answer should be an integer between 0 and 76.

b. In this problem, you will formally prove that the binary exponentiation algorithm is correct.

   (i) (4 marks) Define $s_0 = 1$ and $s_{i+1} = 2s_i + b_{i+1}$ for $0 \le i \le k-1$. Use induction on $i$ to prove that

$$s_i = \sum_{j=0}^{i} b_j 2^{i-j} \quad \text{for } 0 \le i \le k \ .$$

(ii) (4 marks) Let $r_i$, $0 \le i \le k$, be defined as in steps 2 and 3 of the exponentiation algorithm. Use induction on $i$ to prove that $r_i \equiv a^{s_i} \pmod{m}$ for $0 \le i \le k$.

(iii) (2 marks) Prove that $a^n \equiv r_k \pmod{m}$, so the algorithm above does indeed compute $a^n \pmod{m}$ as claimed.

### Problem 4 — A modified man-in-the-middle attack on Diffie-Hellman (10 marks)

Suppose Alice and Bob wish to generate a shared cryptographic key using the Diffie-Hellman protocol. As usual, they agree on a large prime $p$ and a primitive root $g$ of $p$. Suppose also that $p = mq + 1$ where $q$ is prime and $m$ is very small (so $p - 1 = mq$ has a large prime factor, as is generally required). Since $g$ and $p$ are public, it is easy for anyone to deduce $m$ and $q$; for example by successively trial-dividing $p - 1$ by $m = 2, 4, 6, \ldots$ and running a primality test such as the Fermat test on the quotient $q = (p - 1)/m$ until primality of $q$ is established.

Suppose an active attacker Mallory[1] intercepts $g^a \pmod{p}$ from Alice and $g^b \pmod{p}$ from Bob. She sends $(g^a)^q \pmod{p}$ to Bob and $(g^b)^q \pmod{p}$ to Alice.

a. (2 marks) Show that Alice and Bob compute the same shared key $K$ under this attack.

b. (4 marks) Show that there are $m$ possible values for $K$, and that Mallory can compute them all and hence easily guess the correct key $K$ among them.

c. (4 marks) What is the advantage of this variation of the man-in-the-middle attack over the version we discussed in class? Recall that for the attack from class, Mallory simply suppresses the messages $g^a \pmod{p}$ and $g^b \pmod{p}$ between Alice and Bob and replaces them with her own number $g^e \pmod{p}$, which results in the shared key $g^{ae} \pmod{p}$ between Mallory and Alice and the shared key $g^{be} \pmod{p}$ between Mallory and Bob.

### Problem 5 — A simplified password-based key agreement protocol (8 marks)

The following is a simplified (and hence problematic) version of the key generation phase of the password-based key agreement protocol that you are being asked to implement in Problem 9 (the programming problem). Here, a client first performs a one-time registration of their authentication credentials with a server. These credentials can then be used to generate authenticated session keys between server and client via communication over an insecure channel.

All participants agree on a large public prime[2] $N = 2q + 1$, with $q$ prime, and a public primitive root $g$ of $N$. Each client has their own password $p$. To register with the server, a client computes $v \equiv g^p \pmod{N}$ and provides the server with the pair $(I, v)$ where $I$ is the client's user id.[3]

**Protocol:**

1. Client generates a random value $a$ with $0 \le a \le N - 1$, computes $A \equiv g^a \pmod{N}$, and sends $(I, A)$ to server, where $I$ is the Client's user id.

[1] This is a standard name for active attackers and is meant to be reminiscent of the word "malicious".

[2] We denote this prime by $N$, rather than $p$, because the letter $p$ is reserved for the client's password.

[3] In practice, this needs to be done in a secure and tamper-proof manner. Also, in the computation of $v$, the client would use a hash of their password $p$ rather than just $p$. For details, see the protocol description in Problem 9.

Server generates a random value $b$ with $0 \leq b \leq N-1$, computes $B \equiv g^b \pmod{N}$, and sends $B$ to client.

2. Client computes $K_{\text{client}} \equiv B^{a+p} \pmod{N}$.

   Server retrieves client's authentication data $(I, v)$ and computes $K_{\text{server}} \equiv (Av)^b \pmod{N}$.

Note that this protocol is similar to Diffie-Hellman, except that the client's password $p$ and authentication credential $v$ are incorporated in the key computation.

a. [2 marks] Prove that Client and Server have a shared key after executing this protocol, i.e. prove that $K_{\text{server}} = K_{\text{client}}$.

b. [3 marks] Suppose an adversary Mallory obtains client Ian's authentication data $(I, v)$ (by intercepting Ian's transmission to the server upon his registration or by hacking into the server's database). Show how Mallory can masquerade as Ian, i.e. execute the protocol with the server (using a value $A$ of her choice) and generate a valid key $K_{\text{client}}$ that the server believes is shared with Ian.

c. [3 marks] Consider the following two problems:

   - *Key Recovery Problem:* Given any values $A \equiv g^a \pmod{N}$ and $B \equiv g^b \pmod{N}$ and any $v \in \mathbb{Z}_N^*$, find a key $K$ produced by the protocol above.
   - *Diffie-Hellman Problem:* Given any values $A \equiv g^a \pmod{N}$ and $B \equiv g^b \pmod{N}$, find a Diffie-Hellman key $K \equiv g^{ab} \pmod{N}$.

   Note that the exponents $a$ and $b$ are assumed to be unknown for both these problems. Show how an attacker Mallory who can solve any instance of the key recovery problem can solve any instance of the Diffie-Hellman problem. (So informally, breaking the key agreement protocol above is at least as hard as breaking Diffie-Hellman.)

## Written Problems for MATH 318 only

### Problem 6 — Discrete logs with respect to different primitive roots (6 marks)

Let $p$ be a prime and $g$ a primitive root of $p$. Recall that for any $a \in \mathbb{Z}_p^*$, the *discrete logarithm of a with respect to g* is unique integer $x$ with $0 \leq x \leq p-2$ and $g^x \equiv a \pmod{p}$.

Recall that the discrete problem is asserted to be computationally intractable. This raises the natural question of whether this problem might be easier to solve for some primitive roots than for others. In this problem you will prove that that the difficulty of the discrete logarithm problem is independent of the choice of primitive root.

Specifically, let $g, h$ be primitive roots of $p$ and assume that for any element in $\mathbb{Z}_p^*$, computing its discrete logarithm with respect to $g$ is easy. Give an algorithm for easily computing its discrete logarithm with respect to $h$.

## Problem 7 — Primitive roots for safe primes (6 marks)

Let $q \geq 3$ be a prime such that $p = 2q+1$ is also prime. Let $g$ be any primitive root of $p$. Prove that with the exception of $g^q \pmod{p}$, all the odd powers of $g$ (i.e. $g, g^3 \pmod{p}, g^5 \pmod{p}, \ldots, g^{p-2} \pmod{p}$), are primitive roots of $p$.

(*Hint:* the following fact about divisibility, which you may use without proof, might come in handy: for any three nonzero integers $a, b, c$, if $a$ is a divisor of the product $bc$ and $\gcd(a, b) = 1$, then $a$ is a divisor of $c$.)

## Problem 8 — An algorithm for extracting discrete logarithms (25 marks)

Let $p$ be a large prime and $g$ a fixed primitive root of $p$. Let $h \in \mathbb{Z}_p^*$ be the modular inverse of $g$, so $gh \equiv 1 \pmod{p}$. Let $a \in \mathbb{Z}_p^*$. Define the following lists of elements in $\mathbb{Z}_p^*$:

$$y_i \equiv ah^i \pmod{p},\ 0 \leq i \leq m-1;$$

$$z_j \equiv (g^m)^j \pmod{p},\ 0 \leq j \leq k-1.$$

Here, $m$ is a positive integer (an as yet unspecified parameter) and $k$ is the smallest integer with $k \geq (p-1)/m$, so $k \geq (p-1)/m > k-1$.

a. (4 marks) Prove that there exist indices $i, j$ with $0 \leq i \leq m-1$ and $0 \leq j \leq k-1$ such that $y_i = z_j$.
(*Hint:* Division with remainder of $x$ by $m$ where $x$ is the (unknown) discrete logarithm of $a$ with respect to $g$.)

b. (5 marks) Consider the following procedure for computing the discrete logarithm of $a$ with respect to $g$.

1. Compute the list $\mathcal{Y} = (y_0, y_1, \ldots, y_{m-1})$
2. If $y_i \equiv 1 \pmod{p}$ for some $i \in \{0, 1, \ldots, m-1\}$, then output $i$ and quit.
3. Compute the list $\mathcal{Z} = (z_0, z_1, z_2, \ldots, )$ until an element $z_j$ is found that appears in $\mathcal{Y}$.
4. Upon finding such a match, say $z_j = y_i$, output $x \equiv jm + i \pmod{p-1}$.

Prove that this procedure terminates and outputs the discrete logarithm of $a$.

c. (4 marks) The computational cost of this algorithm is determined by the number of modular multiplications required to generate the lists $\mathcal{Y}$ and $\mathcal{Z}$ (we may ignore the cost of step 2 and of searching the list $\mathcal{Y}$ for a match to an element in $\mathcal{Z}$ in step 3 as these tasks take negligible time).

Assume the worst case scenario where the entire list $\mathcal{Z} = (z_0, z_1, \ldots, z_{k-1})$ needs to be generated before a match with an element in the list $\mathcal{Y}$ is found. How many modular multiplications are required to extract the discrete logarithm of $a$ using the procedure above? Your count should be as accurate as possible (i.e. don't count modular multiplications that aren't needed). You may assume that $k$ and $g^m \pmod{p}$ have been precomputed as they are independent of $a$.

d. (5 marks) How should $m$ be chosen to minimize the number of modular multiplication determined in part (c)? Explain your choice.

   (*Hint: Your answer should be a function of $p$ that's close to $\sqrt{p}$.*)

e. Let $p = 107$.

   (i) (2 marks) Use the primitive root test from class to verify that 2 is a primitive root of $p$. Show your work.

   (ii) (4 marks) Use the procedure from part (b) and your choice of $m$ from part (d) to compute the discrete logarithm of 6 with respect to 2 in $\mathbb{Z}_{107}^*$. Show your work. You may want to verify that your final answer is correct.

## Programming Problem for CPSC 418 only

### Problem 9 — Secure password based authentication and key exchange (37 marks)

**Overview.** This problem considers the full, secure version of the password-based key agreement protocol introduced in Problem 5. This protocol, executed by a Client and a Server, allows the Client to demonstrate to the Server knowledge of a password, but neither the password nor any other information that could be used to derive the password need to be transmitted. Additionally, the Server does not store password-equivalent data, so someone who intercepts authentication data or steals them from the Server's database is unable to masquerade as the Client without brute-forcing the Client's password.

To execute the protocol, there is an initialization and registration phase between the Client and Server. Doing this securely is in itself a complex problem, and so we instead perform a simplified version as described below.

**The Server initializes in the following manner:**

1. Generates a 512-bit safe prime $N$ by first generating a random 511-bit prime $q$ and checking whether $N = 2q + 1$ is prime. Repeat until a valid $N$ is found.
2. Finds a primitive root $g$ of $N$, which may be up to 512 bits in size.
3. Computes the quantity $k = H(N||g)$, where '$||$' denotes concatenation and $H$ is a cryptographically secure hash function.
4. Opens a socket connection on a specified IP and port and attempts to receive a single byte, whose value indicates the Server response. This byte is limited to the UTF-8 encoding of the characters '`r`' indicating Client wishes to initiate registration; '`p`' indicating the Client wishes to initiate the protocol; and '`q`' indicating the Client wishes the Server to shut down.[4]

**The Server performs registration as follows:**

1. Server sends the values $N, g$ to the Client.
2. Server receives a single byte containing the length of the Client's username, followed by `username`, $s, v$. Server stores these values.
3. Server resumes listening on the socket, for another command.

[4] A true implementation would not have this last feature, but it makes debugging much easier.

**The Client performs registration as follows:**

1. The username `username` and a password `pw` are retrieved from the command line.[5]
2. Generates a random 16-byte salt[6] $s$.
3. Connect to the Server's socket and sends a one-byte character '`r`';
4. Receives the values $N, g$;
5. Computes $x = H(s||\texttt{pw})$ and $v \equiv g^x \pmod{N}$;
6. Computes and sends a single byte containing the byte-length of `username`, followed by `username` encoded in UTF-8, $s$, and $v$.

In a similar manner, if a Client connects to the Server and wishes to initiate the protocol, they will first send a single byte corresponding to '`p`' before commencing with the protocol described below.

**Protocol.**

To generate and verify a shared authenticated session key, the Server and Client perform the following steps:

1. Client generates a random value $a$ with $0 \leq a \leq N - 1$, computes $A \equiv g^a \pmod{N}$, and sends the length of the username as a single byte, followed by `username`, $A$.

   Server generates a random value $b$ with $0 \leq b \leq N - 1$, computes $B \equiv kv + g^b \pmod{N}$, and sends $s, B$, where $s$ is the Client's salt.
2. Client and Server compute $u \equiv H(A||B) \pmod{N}$.
3. Client computes $K_{\text{client}} \equiv (B - kv)^{a+ux} \pmod{N}$.

   Server computes $K_{\text{server}} \equiv (Av^u)^b \pmod{N}$.
4. Client computes and sends $M_1 = H(A||B||K_{\text{client}})$.
5. Server computes $H(A||B||K_{\text{server}})$ and compares the result to $M_1$. If they do not match, the authentication has failed and the socket is closed.
6. Server computes and sends $M_2 = H(A||M_1||K_{\text{server}})$.
7. Client computes $H(A||M_1||K_{\text{client}})$ and compares the result to $M_2$. If they do not match, the authentication has failed.
8. The Server closes the socket and waits for further connections.

Steps 1-3 generate the authenticated key shared between the Client and the Server. Steps 4-7 verify that both parties have computed the same shared key. If executed honestly, $K_{\text{client}}$ and $K_{\text{server}}$ are equal and the Server and Client were able to authenticate each other and establish a shared session key.

- At the end of either registration or the protocol, the Server should resume listening on the socket.

[5] The program template will do this for you, as well as check that the length of the username encodes to less than 255 bytes.

[6] In cryptography, a *salt* is a random piece of data used as an additional input to a one-way function that hashes data, a password or passphrase.

- Note that in order to reliably receive information from the socket, the byte length of the data must typically be known. Unless otherwise specified, the size of any value taken modulo $N$ should be the same length as $N$ (up to 512 bits). Since SHA2-256 is used, any hashed value that is not taken modulo $N$ is therefore 256 bits in size. The encoded username may be up to 255 bytes in length.
- Please note the importance of the order that values are sent over the socket.

**Problem.** Your task is to complete the template program named

`basic_auth.py`

which performs the above password-based key agreement protocol. The program consists of functions corresponding to establishing a connection and transmitting data through a socket, parameter generation, and the actions performed by the Client and the Server.

All messages over the socket should be echoed to standard output by both the sending and receiving party. Each echoed message should *clearly* indicate its sender and intended receiver. Use a template like the following:[7]

```
Server:  N = (integer value of N)
Client:  Received <(hex representation of incoming data)>
Server:  Authentication was successful.
Client:  ERROR, the socket was closed.
```

For this exercise the hash function $H$ will be SHA2-256 as implemented in the `cryptography` library. Additionally, when randomness is required use either `os.urandom` or the `secrets` library. We will allow use of the `sympy` library, as it is useful for handling prime numbers; however, you are expected to implement the function `prim_root` on your own, and **you may not use sympy's** `primitive_root()` method in your implementation.

**Specifications.** Fill in the empty functions in the template program `basic_auth.py`. Once complete, you should be able to perform both registration and key agreement between two parties, one acting as the `Client` and the other as `Server`, by running

```
python3 basic_auth.py --server 127.0.4.18:3180
python3 basic_auth.py --client 127.0.4.18:3180
python3 basic_auth.py --quit 127.0.4.18:3180
```

You may also combine these actions with one invocation of `basic_auth.py`, although this makes debugging substantially more difficult. There will be additional command-line options to change the username and password from the defaults.

In detail, the functions you'll need to fill in fall into roughly three categories:

a. Networking Functions:

(i) `create_socket(...)`, which creates a socket connection on the specified IP and port.

[7] The printed text is primarily for human eyes, not computer parsing, so there is no need to match these examples precisely.

(ii) `send(...)`, which sends bytes through the specified socket.
(iii) `receive(...)`, which receives a specified bytes through the socket.

b. Low-level Functions:

(i) `safe_prime(...)`, which creates a random safe prime $N$ of a specified bit-length and whose most significant bit is 1.
(ii) `prim_root(...)`, which finds a primitive root $g$ of $N$ for some prime $N$.

**Note: You may not use sympy's `primitive_root` method for your implementation.**

(iii) `calc_x(...)`, which computes the value $x$ from the registration step using the salt and password.
(iv) `calc_u(...)`, which computes the value $u$ using the provided inputs.
(v) `calc_A(...)`, which computes the value $A$ using the provided inputs.
(vi) `calc_B(...)`, which computes the value $B$ using the provided inputs.
(vii) `calc_K_client(...)`, which computes the value $K_{client}$.
(viii) `calc_K_server(...)`, which computes the value $K_{server}$.
(ix) `calc_M1(...)`, which computes the value $M_1$.
(x) `calc_M2(...)`, which computes the value $M_2$.

c. High-level Functions:

(i) `client_register(...)`, which sends and receives information to and from the Server as described in the Client registration.
(ii) `server_register(...)`, which receives and sends information from and to the Client through the socket connection, as outlined in the Server registration.
(iii) `client_protocol(...)`, which performs the Client's side of the protocol.
(iv) `server_protocol(...)`, which performs the Server's side of the protocol.
(v) `client_prepare(...)`, which computes the Client's registration tuple $(username, s, v)$.
(vi) `server_prepare(...)`, which computes the Server's registration values $N, g, k$.

Note that some values may be supplied as an integer or as a `bytes` object; your functions will need to translate between them as the context requires. All numbers are converted to `bytes` via network byte order, which is big-endian.[8] Additional details and documentation of these functions can be found in the template program found on the Piazza resources page.

**Submission.** Submit a completed version of the template program with filename

`basic_auth.py`

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

Provide a description of your implementation 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 should include the following:

- The procedures you used for generating your prime $N$ and your primitive root $g$ of $N$.
- 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.

[8] Functions will be supplied with the template to help with conversion.