CPSC 418 / MATH 318 — Introduction to Cryptography

ASSIGNMENT 3

Due: **Wednesday, Apr. 14, 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 & 7 are worth 37 marks and are for MATH 318 students only; CPSC 418 may do these for limited extra credit.

Problem 8 is worth 37 marks and is 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 — Flawed MAC designs (11 marks)

For this problem, you need to carefully trace through the given MAC algorithms, and specifically through the underlying iterated hash function, to understand the given attacks and explain how computation resistance is defeated.

Recall that iterated hash functions employ a *compression function* f in multiple rounds; SHA-1 is an example of such a hash function. Here, f takes as input pairs of n-bit blocks and outputs n-bit blocks, for some $n \in \mathbb{N}$. The compression function f is assumed to be public, i.e. anyone can compute values of f. As a result, the hash function is also public, so any one can compute hash values. For simplicity, we assume that the bit length of any message to be hashed is a multiple of n, i.e. messages have been padded appropriately prior to hashing. Then the input to an iterated hash function is a message M consisting of L blocks $P_1, P_2, \ldots, P_L$, each of length n. An algorithmic description of an n-bit iterated hash function is given as follows (as usual. "$\|$" denotes concatenation of strings).

Algorithm ItHash

Input: A message $M = P_1\|P_2\|\cdots\|P_L$
Output: An n-bit hash of M
1: Initialize $H := 0^n$ (n zeros)
2: for $i = 1$ to L do
3: $H := f(H, P_i)$
4: end for
5: Output H

This problem demonstrates that the two "obvious" designs for using an iterated hash function as the basis for a MAC — prepending or appending the key to the message and then hashing — are not secure. You will prove this by mounting specific attacks that defeat computation resistance. For simplicity, assume that keys also have length n, the same as the message block length.[1]

a. (5 marks) Define a message authentication function PHMAC ("Prepend Hash MAC") via

$$\text{PHMAC}_K(M) := \text{ItHash}(K\|M) = \text{ItHash}(K\|P_1\|P_2\|\cdots\|P_L)$$

for any message $M = P_1\|P_2\|\cdots\|P_L$.

Suppose an attacker knows a message/PHMAC pair $(M_1, \text{PHMAC}_K(M_1))$. Let X be an arbitrary n-bit block and put $M_2 = M_1\|X$. Show how an attacker can compute $\text{PHMAC}_K(M_2)$ without knowledge of K, thereby defeating computation resistance for PHMAC.

Hint: Let $M_1 = P_1\|P_2\|\cdots\|P_L$. Tracing through the ItHash algorithm, compare the results of the first $L+1$ rounds of $\text{ItHash}(K\|M_1)$ and $\text{ItHash}(K\|M_2)$. Then explain how the attacker can compute $\text{PHMAC}_K(M_2)$ from $\text{ItHash}(K\|M_2)$ without knowledge of K.

[1] The attack in part (a) will in fact work for any key length, but for part (b), this key length restriction is required.

b. (6 marks) Define a message authentication function AHMAC_K ("Append Hash MAC") via

$$\text{AHMAC}_K(M) := \text{ItHash}(M\|K) = \text{ItHash}(P_1\|P_2\|\cdots\|P_L\|K)$$

for any message $M = P_1\|P_2\|\cdots\|P_L$.

Assume that ItHash is not weakly collision resistant, and suppose an attacker knows a message/AHMAC pair $(M_1, \text{AHMAC}_K(M_1))$. Show how she can find (without knowledge of K) a second message/AHMAC pair $(M_2, \text{AHMAC}_K(M_2))$, thereby defeating computation resistance.

Hint: Note that on input any L-bit message, the first L rounds of the computation of $\text{AHMAC}_K(M)$ do not depend on K, just on M.

Problem 2 — Fast RSA decryption using Chinese remaindering (7 marks)

In this problem, as usual, a user Alice has an RSA public key (e, n) with corresponding private key d. Here, $n = pq$ for distinct large primes p and q.

If Alice does not discard p and q after computing n and $\phi(n)$, she can employ an alternative decryption procedure as described below (based on the *Chinese Remainder Theorem* which some of you may have seen before). For a given ciphertext C $(1 \le C \le n-1, \gcd(C, n) = 1)$, she proceeds as follows:

Step 1. Compute

$$\begin{aligned} d_p &\equiv d \pmod{p-1}, \quad 0 \le d_p \le p-2\,, \\ d_q &\equiv d \pmod{q-1}, \quad 0 \le d_q \le q-2\,. \end{aligned}$$

Step 2. Compute

$$\begin{aligned} M_p &\equiv C^{d_p} \pmod{p}, \quad 1 \le M_p \le p-1\,, \\ M_q &\equiv C^{d_q} \pmod{q}, \quad 1 \le M_q \le q-1\,. \end{aligned}$$

Step 3. Use the Extended Euclidean Algorithm to find integers x, y such that

$$px + qy = 1\,.$$

(Such integers exist because $\gcd(p, q) = 1$.)

Step 4. Set $M \equiv pxM_q + qyM_p \pmod{n}$, $0 \le M \le n-1$.

Prove that if the above procedure decrypts correctly. That is, if $C \equiv M^e \pmod{n}$ is a ciphertext obtained by encrypting a message M the "normal" RSA way, and M' is the result of applying the procedure above to C, prove that $M' = M$.

Hint: You may use without proof the fact that $a \equiv b \pmod{n}$ if and only if $a \equiv b \pmod{p}$ and $a \equiv b \pmod{q}$ for any $a, b \in \mathbb{Z}$. The "only if" direction follows directly from the definition of congruence; the "if" direction follows from the Chinese Remainder theorem.

Remark: This method performs two modular exponentiations for decryption (in step 2), as opposed to just one modular exponentiation for the ordinary RSA decryption procedure. However, the moduli p and q have only about half the bit length of n, and the exponents d_p and d_q generally have about half the bit length of d (as $d \lessapprox n$, $d_p \lessapprox p$ and $d_1 \lessapprox q$). Since the computational effort of steps 1, 3 and 4 is negligible compared to step 2, Chinese remainder decryption is generally almost four times as fast as ordinary RSA decryption. So this is what is generally used in practice.

Problem 3 — RSA primes too close together (21 marks)

This problem explores a *difference of squares* approach to factoring an RSA modulus due to Fermat, which is hence also known as *Fermat factorization*. Fermat's idea was to attempt to find a representation of an integer n as a difference of squares $n = x^2 - y^2$ where $0 < y < x < n$, which leads to a factorization $n = (x + y)(x - y)$. When n is an RSA modulus whose prime factors are very close together, the quantity y is very small compared to x and we will see that this gives rise to a serious factoring attack on RSA.

Let $n = pq$ with odd primes p, q, and assume without loss of generality that $p > q$. As always, all square roots are positive, i.e. when we write $\sqrt{z}$ for some $z > 0$, we mean the positive square root of z.

a. (4 marks) Prove that if x, y are integers with $x > y > 0$ and $n = x^2 - y^2$, then

$$x = \frac{n+1}{2}\,,\ y = \frac{n-1}{2} \quad \text{or} \quad x = \frac{p+q}{2}\,,\ y = \frac{p-q}{2}\ . \tag{1}$$

Note that it is not enough to prove that these given values for x and y satisfy $n = x^2 - y^2$. The point is to prove that there are no integer values for x and y other than the ones given here.

b. (2 marks) Prove that $p + q < n + 1$.

c. (3 marks) Use the inequality $p > q$ to prove that $\sqrt{n} < \dfrac{p+q}{2} < p$.

d. (4 marks) The idea for factoring n is to iterate over integers x that are potential candidates for $(p + q)/2$, starting with the smallest integer exceeding $\sqrt{n}$ (the lower bound from part (c)) and searching in increments of 1 until a value x is found for which $y := \sqrt{x^2 - n}$ is an integer. The assertion (which requires proof!) is that $x = (p + q)/2$ and $y = (p - q)/2$, in which case the method returns the factor $x - y = q$. Here is pseudocode for this factorization algorithm:

Algorithm Fermat Factorization

```
Input: n = pq with p > q
Output: q
1: Put x = ⌈√n⌉  (√n rounded up to the nearest integer)
2: y := √(x^2 − n)
3: while y is not an integer do
4:     x := x + 1;  y := √(x^2 − n)
5: end while
6: Output x − y.
```

Use parts (a)-(c) to prove that this algorithm terminates s soon as $x = (p + q)/2$ and outputs q. Note that there are three things to show here:

- The "while" condition is satisfied when $x = (p + q)/2$;
- $x = (p + q)/2$ is the first value that satisfies the "while" condition;
- The algorithm outputs q.

e. (2 marks) Show that the "while" condition in step 3 is tested $x - \lceil \sqrt{n} \rceil + 1$ times, where $x = (p+q)/2$.

f. (4 marks) Prove that $x - \lceil \sqrt{n} \rceil < \dfrac{y^2}{2\sqrt{n}}$ where $x = (p+q)/2$ and $y = (p-q)/2$.

Hint: Find a suitable upper bound on $(x - \sqrt{n})(x + \sqrt{n})$ and divide by $x + \sqrt{n}$. Prove that the result yields a bound on $x - \lceil \sqrt{n} \rceil$.

g. (2 marks) Finally, the coup-de-grâce. Suppose $p - q < 2B\sqrt[4]{n}$ for some integer B that is very small compared to n; e.g. B could be on the order of a power of $\log_2(n)$ or even a constant number. In other words, p and q are very close together; they agree in nearly half of their most significant bits. Prove that the above algorithm factors n after at most

$$\frac{B^2}{2} + 1$$

steps, where a *step* corresponds the an evaluation of the "while" condition in step 3.

Problem 4 — El Gamal is not semantically secure (12 marks)

This problem requires typesetting Legendre symbols. To facilitate this, include at the beginning of your assignment file, right after the line `\documentclass{...}` the two lines

```
\usepackage{amsmath}
\providecommand{\Leg}[2]{\genfrac{(}{)}{}{}{#1}{#2}}
```

Be sure that you copy these *verbatim*; the easiest is to copy and paste them right from this PDF file. The assignment template provided on the Piazza Resourcesd page already includes these lines. The command `$\Leg{a}{n}$` will produce the typeset output $\left(\frac{a}{n}\right)$, which is much easier than producing a fraction with large parentheses around it.

Recall that for the El Gamal public key cryptosystem, a user Alice produces her public and private keys as follows:

Step 1. Selects a large prime p and a primitive root g of p.

Step 2. Randomly selects x such that $0 < x < p - 1$ and computes $y \equiv g^x \pmod{p}$.

Alice's public key is $\{p, g, y\}$

Alice's private key is $\{x\}$

To encrypt a message $M \in \mathbb{Z}_p^*$ intended for Alice, Bob selects a random integer $k \in \mathbb{Z}_{p-1}$, computes $C_1 \equiv g^k \pmod{p}$ and $C_2 \equiv My^k \pmod{p}$, and sends $C = (C_1, C_2)$ to Alice.

Alice decrypts the ciphertext $C = (C_1, C_2)$ by computing $C_2 C_1^{p-1-x} \equiv M \pmod{p}$.

In this problem, you will prove that the El Gamal public key cryptosystenm is not polynomially secure, and hence not semantically secure. This is because an attacker Mallory can distinguish messages according to whether they are quadratic residues or quadratic nonresidues modulo p.

Mallory mounts her attack with the following procedure:

Step 1. Selects two messages M_1 and M_2 such that $M_1 \in QR_p$ and $M_2 \in QN_p$, and obtains the ciphertext $C = (C_1, C_2) = E(M_i)$ where $i = 1$ or 2.
(Mallory's task is precisely to ascertain whether $i = 1$ or $i = 2$.)

Step 2. Computes the Legendre symbols $\left(\frac{y}{p}\right)$, $\left(\frac{C_1}{p}\right)$ and $\left(\frac{C_2}{p}\right)$.

Step 3. If $\left(\frac{y}{p}\right) = 1$ and $\left(\frac{C_2}{p}\right) = 1$, she asserts that $C = E(M_1)$.

If $\left(\frac{y}{p}\right) = 1$ and $\left(\frac{C_2}{p}\right) = -1$, she asserts that $C = E(M_2)$.

If $\left(\frac{y}{p}\right) = -1$, $\left(\frac{C_1}{p}\right) = 1$ and $\left(\frac{C_2}{p}\right) = 1$, she asserts that $C = E(M_1)$.

If $\left(\frac{y}{p}\right) = -1$, $\left(\frac{C_1}{p}\right) = 1$ and $\left(\frac{C_2}{p}\right) = -1$, she asserts that $= E(M_2)$.

If $\left(\frac{y}{p}\right) = -1$, $\left(\frac{C_1}{p}\right) = -1$ and $\left(\frac{C_2}{p}\right) = 1$, she asserts that $C = E(M_2)$.

If $\left(\frac{y}{p}\right) = -1$, $\left(\frac{C_1}{p}\right) = -1$ and $\left(\frac{C_2}{p}\right) = -1$, she asserts that $C = E(M_1)$.

Note that this procedure requires three Legendre symbol computations — which can be done with modular exponentiation by Euler's Criterion — and hence always takes polynomial time. Note also that Mallory states her assertions with certainty, i.e. probability 1.

Prove that Mallory's assertions are correct, so the El Gamal system is not semantically secure.

Problem 5 — An IND-CPA, but not IND-CCA secure version of RSA (12 marks)

Consider the following semantically secure variant of the RSA public key cryptosystem:

Parameters:

- m — length of plaintext messages to encrypt (in bits)
- (n, e) — Alice's RSA public key (n has k bits)
- d — Alice's RSA private key
- $H : \{0,1\}^k \mapsto \{0,1\}^m$ a public random function

Encryption of an m-bit message $M \in \mathbb{Z}_n*$:

Step 1. Generate a random k-bit number $r < n$.

Step 2. Compute $C = (s \| t)$ where $s \equiv r^e \pmod{n}$ and $t = H(r) \oplus M$.

Decryption of a ciphertext $C = (s \| t)$:

Step 1. Separate C into s and t.

Step 2. Compute $M \equiv H(s^d \pmod{n}) \oplus t$.

Prove that this cryptosystem is not IND-CCA secure.

Hint: To mount her CCA, Mallory gets to choose two plaintexts and submit a ciphertext C' for decryption. Almost any two plaintexts M_1, M_2 will do. Let

$$C = (s \| t) = (r^e \pmod{n} \parallel H(r) \oplus M_i) \quad \text{where } i = 1 \text{ or } 2$$

be the encryption of one them; Mallory needs to ascertain whether $i = 1$ or $i = 2$. Mallory chooses the ciphertext $C' = (s \| t \oplus M_1)$ and obtains its decryption (make sure that $C' \neq C$; that's a requirement in the CCA). Now trace through the decryption method applied to C and use the result to figure out how Mallory can detect whether C is the encryption of M_1 or M_2. (Of course Mallory can't actually decrypt C, but she/you can still apply the decryption method symbolically to C.)

Written Problems for MATH 318 only

Problem 6 — A primality test of sorts (12 marks)

Let N be an integer with$N > 1$. In this problem, you will prove the assertion

$$(N-1)! \equiv -1 \pmod{N} \text{ if and only if } N \text{ is prime} \tag{2}$$

and analyze its suitability for primality testing.

For the primes $N = 2$ and $N = 3$, it is easy to verify that (2) holds, so assume henceforth that $N \geq 4$. Note that in this case, 1 and -1 are distinct elements in $\mathbb{Z}_N$ as $1 \equiv -1 \pmod{N}$ implies that N divides $1 - (-1) = 2$, forcing $N = 2$.

a. (3 marks) Suppose N is prime and let g be primitive root of N. Prove that

$$g^{(N-1)/2} \equiv -1 \pmod{N} .$$

You may use without proof the fact if a prime divides the product of two non-zero integers, then it divides one of the factors.

b. (3 marks) Suppose N is prime. Prove that $(N-1)! \equiv -1 \pmod{N}$.

Hint: Primitive roots. Remember that for any primitive g of N, the set $\mathbb{Z}_N^*$ consists precisely of the powers $g^k \pmod{N}$ with$0 \leq k \leq N-2$.

c. (3 marks) Suppose N is composite. Prove that $(N-1)! \not\equiv -1 \pmod{N}$.

Hint: Consider $(N-1)!$ modulo some prime divisor of N and draw the necessary conclusion about $(N-1)! \pmod{N}$.

d. (3 marks) By parts (b) and (c), the congruence $(N-1)! \equiv -1 \pmod{N}$ can be used as a primality test, actually even as a primality *proof*. Do you think this is a good primality test? How does this compare, for example, to trial division (dividing N successively by $2, 3, 4, \ldots$)? Give a yes/no answer and a concise, coherent and convincing explanation of your answer.

Remark: A wrong answer is "No because $(N-1)!$ is a really big number." This is true, but we are doing modular arithmetic here. The intermediate operands in the computation of $(N-1)! \pmod{N}$ can be kept bounded by reducing modulo N in each step, i.e. by computing $F_{N-1} \equiv (N-1)! \pmod{N}$ via the recurrence $F_0 = 1$ and $F_n \equiv nF_{n-1} \pmod{N}$ for $1 \leq n \leq N-1$.

Problem 7 — An attack on RSA with small decryption exponent (25 marks)

This problem explores an attack on RSA with small private key d.

Preliminaries. Let r be a positive rational number and write $r = a/b$ with $a, b \in \mathbb{N}$. Let $q_0, q_1, \ldots q_m \in \mathbb{N}$ be the sequence of quotients produced by applying the Euclidean Algorithm to the numerator and denominator of r:

$$\begin{aligned}
a &= q_0 b + r_0\,, & 0 < r_0 < b\,,\\
b &= q_1 r_0 + r_1\,, & 0 < r_1 < r_0\,,\\
r_0 &= q_2 r_1 + r_2\,, & 0 < r_2 < r_1\,,\\
&\vdots\\
r_{m-3} &= q_{m-1} r_{m-2} + r_{m-1}\,, & r_{m-1} = \gcd(a,b)\,,\\
r_{m-2} &= q_m r_{m-1} + r_m\,, & r_m = 0\,.
\end{aligned}$$

Recall the familiar sequences

$$\begin{array}{cccccccc}
A_{-2} & = & 0, & A_{-1} & = & 1, & A_i & = & q_i A_{i-1} + A_{i-2} \quad \text{for } 0 \le i \le m,\\
B_{-2} & = & 1, & B_{-1} & = & 0, & B_i & = & q_i B_{i-1} + B_{i-2} \quad \text{for } 0 \le i \le m.
\end{array}$$

The quotients A_i/B_i $(0 \le i \le m)$ are called the *convergents* of r because they oscillate around and converge toward r as i increases, with $A_m = a$, $B_m = b$, and hence $A_m/B_m = r$. In fact, the following theorem (which you may use without proof) asserts that any rational number sufficiently close to r must occur as one of the convergents:

Theorem. Let $r = \dfrac{a}{b} \in \mathbb{Q}$ with $a, b > 0$, and let $\dfrac{A}{B} \in \mathbb{Q}$ be a fraction in lowest terms such that $\left| r - \dfrac{A}{B} \right| < \dfrac{1}{2B^2}$. Then $A = A_i$ and $B = B_i$ for some $i \in \{0, 1, \ldots, m\}$.

Now back to RSA. Let $n = pq$ where p, q are odd primes satisfying

$$q < p < 2q\ .$$

These inequalities are reasonable as p and q are usually assumed to have the same bit length. Let e, d be integers with $1 < e, d < \phi(n)$ and $ed \equiv 1 \pmod{\phi(n)}$. Let $k \in \mathbb{Z}$ satisfy $ed = 1 + k\phi(n)$ and suppose that d is small compared to n; specifically,

$$d < \frac{\sqrt[4]{n}}{\sqrt{6}}. \tag{3}$$

a. (5 marks) Prove that $1 \le k < d$ and $\gcd(d, k) = 1$.

b. (3 marks) Prove that $2 \le n - \phi(n) < 3\sqrt{n}$.

c. (4 marks) Use parts (a) and (b) to prove that $0 < kn - ed < 3d\sqrt{n}$.

d. (4 marks) Conclude from part (c) and the upper bound (3) on d that $0 < \dfrac{k}{d} - \dfrac{e}{n} < \dfrac{3}{\sqrt{n}} < \dfrac{1}{2d^2}$.

e. (3 marks) Let $q_0, q_1, \ldots, q_m$ be the quotients obtained when applying the Euclidean algorithm to e and n, and let A_i, B_i be the associated sequences as defined above. Use the theorem from above to prove that $k = A_i$ and $d = B_i$ for some $i \in \{0, 1, \ldots, m\}$.

f. (6 marks) Use part (e) to devise a procedure for finding d efficiently. Explain why your procedure works and why it is efficient, i.e. argue that the number of computation steps performed by your procedure is small.

Programming Problem for CPSC 418 only

Problem 8 — Secure file transfer with prior key agreement (37 marks)

Don't be daunted by the long description of this problem! Most of it is very clear specifications, including those for the autograder, to make your life easier.

Overview. Transport Layer Security (TLS) is a security protocol which aims to provide end-to-end communication security over networks. It provides both privacy and data integrity. TLS has many steps, but our focus will be the *cipher suite*, a set of algorithms that add cryptographic security to a network connection. There are four main components to a cipher suite:

- Key Exchange Algorithm
- Authentication Algorithm (signature)
- Bulk Encryption Algorithm (block cipher and mode of operation)
- Message Authentication Algorithm

Cipher suites are specified in shorthand by a string such as

SRP-SHA3-256-RSA-AES-256-CBC-SHA3-256.

- This implies SRP-SHA3-256 as the key exchange algorithm, RSA signatures for authentication, AES-256-CBC for encryption and SHA3-256 as the MAC (used as HMAC).
- SRP is the protocol implemented in Assignment 2. The hash function used in the protocol is specified as SHA3-256.
- SHA3-256 as the MAC implies the use of HMAC with SHA3-256.

In a *TLS handshake*, upon connection two parties automatically negotiate TLS settings, including the cipher suite to use. Through an exchange of messages the two parties verify each other and establish a session key. In this assignment, the two parties will be a server and client. The server will authenticate itself to the client by means of a *certificate*, and if successful, will derive a shared session key. The two parties are then able to use the session key to exchange encrypted and authenticated messages.

In reality, a certificate is an electronic document which contains a public key and information about the owner of the key. The certificate must be signed by a *trusted authority* to verify the contents of the certificate. For us, the certificate will merely be a *name* concatenated with a *public key*, concatenated with the signature of a *trusted third party* (TTP).

For this assignment, your task is to implement a toy version of the TLS handshake using the cipher suite specified above.[2] The protocol will have two parties a Client and Server, which rely on a certificates provided by a TTP.

To execute the protocol, the Client and Server first need to obtain certificates from the TTP. We perform a simplified version of this process as follows:

The TTP initializes in the following manner:

- Generates two RSA primes p and q of size 512-bits and computes $N = pq$.
- Generates its own RSA key-pair (N, e) and (N, d).
- Opens a socket connection on a specified IP and port and attempts to receive a single byte, whose value indicates the TTP response. This byte is limited to the UTF-8 encoding of the characters '`s`' indicating the connecting party wishes the TTP to sign their data; '`k`' indicating the connecting party wishes to obtain the TTP public key; and '`q`' indicating the party wishes the TTP to shut down.[3]

The TTP performs signing as follows:

a. Receives the following: a single byte encoding of the length of *name*, a utf-8 encoded *name* and an RSA public key (N, e) where N, e are each encoded as byte-arrays of size 128.

b. Hashes the byte array $(name||N||e)$ using SHA3-512 to obtain a 512-bit value t, stores it, and hashes t once again to produce t'.

c. Taking the byte array $t||t'$, interpret this as a big-endian integer and reduce by the TTP's RSA modulus to get the value S. [4]

d. Compute TTP_SIG, the RSA signature on S.

e. Send the values $(N', \text{TTP\_SIG})$ to the connected party, where N' is the TTP's RSA modulus.

f. Resume listening on the socket for another command.

The TTP performs key-request as follows:

a. Sends its public key (N, e).

b. Resumes listening on the socket connection to receive another command.

The Server performs signature-request as follows:

a. Connect and send a one-byte character '`s`' to the TTP.

b. Send a single byte encoding of the length of *name*, a utf-8 encoded *name* and an RSA public key (N, e) where N, e are each encoded as byte-arrays of size 128.

c. Receive and store the TTP's N and TTP_SIG.

[2] With TLS 1.3, there has been a move towards using *authenticated encryption schemes* such as AES-GCM. For pedagogical reasons, we will look at a more classic cipher-suite.

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

[4] The purpose of this is to make full use of the RSA message space.

d. Close the socket connection.

The Client performs key-request as follows:

a. Connect and send a one-byte character '`k`' to the TTP.

b. Receive and store the TTP's N and d.

c. Close the socket connection.

The following steps must be added to the Server initialization from the previous assignment before opening a listening socket:

a. Obtain a name, `server_name`.

b. Generates two RSA primes p' and q' of size 512-bits and computes $N' = p'q'$.

c. Generates an RSA key-pair (N', e') and (N', d').

d. Connect and request a signature from the TTP

Once the Client and Server have both obtained the relevant information from the TTP and the Client has registered with the Server, the TLS handshake proceeds (finally!) as follows:

a. Client sends the length of its username as a single byte, followed by `username` to the Server.

b. The Server sends a *certificate* consisting of the server's name and public key as well as a signature of these values by the TTP.

- The server sends the length of `server_name` as a single byte, followed by

$$\texttt{server\_name} \; ||N'||e'||\text{TTP\_SIG}$$

  where $N', e,'$ TTP_SIG are each encoded as 128-byte arrays.
- Upon receiving the certificate, the Client should *verify* the signature of the TTP. If verification fails, the client should close the socket.

c. Initiate the protocol from Assignment 2 with the following modifications:

- Instead of the Client sending A as plaintext, they will send $Enc(A)$, the RSA encryption of A under the Server's public key.
- Upon receiving $Enc(A)$ the server must decrypt to obtain A.
- In case you did not implement the following check in Assignment 2: the server should ensure that the value A is not congruent to 0 under the SRP modulus and abort otherwise (it may be fun to think about why).
- The remainder of the protocol proceeds as n Assignment 2 (derive the shared key and verify).

d. With the shared key derived, take the first 32-bytes as key for AES-256. Use the next 32 bytes as the key for HMAC.

e. Pad the file if necessary, then encrypt it using the appropriate key.

f. Also note that you will require an IV for AES-CBC. This should be randomly generated and prepended to the encrypted message (just as in Assignment 1). The IV is considered part of the ciphertext.

g. Generate a tag of the ciphertext bytes using HMAC-SHA3-256 with the appropriate key, and append the tag to the encrypted bytes.[5]

h. Store the length of the ciphertext in a 4-byte array, then send the length followed by the ciphertext to the Server.

i. Have the server receive the ciphertext and decrypt and verify the tag.

Problem Your task is to complete the template program

`basic_handshake.py`

which performs the above handshake 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, Server and TTP.

It is recommended to echo messages sent over the socket to standard output, as this makes debugging the protocol substantially easier. This is not required, but in prior years students benefited greatly from being able to watch the network traffic. The `varprint(...)` function is useful for this.

In adherence to the specified TLS ciphersuite, the hash function H used in the protocol will be **SHA3-256** as implemented in the `cryptography` library. **Remember to change this when using the protocol from Assignment 2.** Additionally, when randomness is required use either `os.urandom` or the `secrets` library.

We will allow the use of the `sympy` library, as it is useful for handling prime numbers; however, certain functions will not be permitted including

- `sympy.primitive_root()`

Requirements.

a. You must implement your own version of the *RSA related functions. You may not use the built in functions related to RSA from the cryptography library for this exercise.*

b. For efficiency we have chosen very small parameter sizes: The server and TTP choose RSA primes p and q to be 512 bit safe primes. Thus the modulus N will be 1024 bits (128 bytes). See the course notes for the specifics of the RSA encryption and signature schemes.

c. All other primitives should make use of the `cryptography` library.

d. Please be careful to note the changes in functions used in past assignments. In particular the use of AES-256, SHA3-256, and HMAC in the symmetric protocol.

Specifications. Fill in the empty functions in the template program `basic_handshake.py`. Since this is built on top of the previous assignment, once complete, you should be able to perform the full TLS handshake between two parties, one acting as the `Client` and the other as `Server`, using a `TTP` by running

[5]There is some debate within the cryptography over whether it is more secure to encrypt the hash or append the hash to the encrypted bytes. Since we used the former method on Assignment 1, we will use the latter on this one.

```
python3 basic_handshake.py --ttp
python3 basic_handshake.py --server
python3 basic_handshake.py --client
python3 basic_handshake.py --quit_server --quit_ttp
```

You may also combine these actions with one invocation of `basic_handshake.py`, although this makes debugging substantially more difficult. There will be additional command-line options to change the username and password from the defaults, as well as the IP address and ports the TTP and server listen on.

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

a. Low-level Functions:

(i) `RSA_key.keypair(...)`, which generates e and d for the given p and q.
(ii) `RSA_key.modulus(...)`, which generates p and q for the specified bit length.
(iii) `RSA_key.sign(...)`, which signs x for a given N and d.
(iv) `RSA_key.encrypt(...)`, which encrypts x for a given N and e.
(v) `RSA_key.decrypt(...)`, which decrypts x for a given N and d.
(vi) `pad_encrypt_then_HMAC(...)`, which handles the named operations given a plaintext and two keys.
(vii) `decrypt_and_verify(...)`, which reverses the operations performed by `pad_encrypt_then_HMAC(...)` while verifying the input could be decrypted.

b. High-level Functions:

(i) `ttp_prepare(...)`, which computes the TTPs RSA key pair.
(ii) `ttp_sign(...)`, which signs the connecting party's RSA key.
(iii) `ttp_sendkey(...)`, which sends the TTPs public key.
(iv) `sign_request(...)`, which requests a signature from the TTP.
(v) `key_request(...)`, which requests the TTP's public key.
(vi) `client_protocol(...)`, which performs the Client's side of the protocol, modified from Assignment 2.
(vii) `server_protocol(...)`, which performs the Server's side of the protocol, modified from Assignment 2.
(viii) `server_prepare(...)`, which computes the Server's registration values N, g, k, as well as the Server's RSA key pair.

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.[6] 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_handshake.py`

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

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:

a. A list of files submitted that pertain to the problem and a short description of each file.
b. A list of what is implemented in the event that you are submitting a partial solution or a statement that the problem is fully solved.
c. A list of what is not implemented in the event that you are submitting a partial solution,
d. A list of known bugs or a statement that there are no known bugs.

You may either use your own code or the solution files from Assignments 1 and 2.