Problem Set 3

This problem set is due on *Monday, March 25, 2019* at **11:59 PM**. Please note our late submission penalty policy in the course information handout. Please submit your problem set, in PDF format, on Gradescope. *Each problem should be in a separate PDF.* When submitting the problem in Gradescope, ensure that **all your group members are listed on Gradescope**, and not in the PDF alone.

You are to work on this problem set in groups. For problem sets 1, 2, and 3, we will randomly assign the groups for the problem set. After problem set 3, you are to work on the following problem sets with groups of your choosing of size three or four. If you need help finding a group, try posting on Piazza or email `6.857-tas@mit.edu`. You don't have to tell us your group members, just make sure you indicate them on Gradescope. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration.

Homework must be submitted electronically! Each problem answer must be provided as a separate pdf. Mark the top of each page with your group member names, the course number (6.857), the problem set number and question, and the date. We have provided templates for LaTeX and Microsoft Word on the course website (see the *Resources* page).

Grading: All problems are worth 10 points.

With the authors' permission, we may distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this in your profile on your homework submission.

Our department is collecting statistics on how much time students are spending on psets, etc. For each problem, please give your estimate of the number of person-hours your team spent on that problem.

Problem 3-1. Prime Generation

We've talked in class about various algorithms and techniques that use primes and finite fields. In this problem, we'll implement some of the basic operations and primitives over finite fields.

We've included a Python skeleton, but feel free to use whatever language you're most comfortable with. You'll need arbitrary-precision integers, so you'll either have to write your own library, or use a language that supports them (e.g. Python, Sage or Java with `java.math.BigInteger`).

Many of these operations are implemented in built-in libraries; please don't use these or other libraries. We *do* expect you to use:

- Arbitrary-precision integers, with addition, subtraction, multiplication, division, remainder/modulo, and other basic arithmetic. *Don't* use built-in functions for checking primality or doing modular exponentiation, such as Python's `pow` function with 3 arguments.
- Random number generation, including generating a uniformly random integer in a range (such as `random.randint` in Python). *Don't* use built-in functions for generating a random prime. You don't have to use or write a cryptographically secure random number generator, though that would be a good idea for a real system.

A good rule of thumb is that, if using the library function makes the problem trivial, you probably shouldn't use the library.

Please submit a copy of your code along with your writeup.

Let's begin! We'll be aiming for a very weak security level, so we'll be working with numbers that are around 128 bits long (around 2^{128}). Your computations should be able to finish in well under a minute.

(a) Modular Exponentiation We often need to take exponents a^b modulo a number m. Unfortunately, b can be large (as large as m), so multiplying b times isn't fast enough. Implement fast modular exponentiation via exponentiation-by-squaring.

(b) Random Prime Generation Our next task is to generate a random b-bit prime (in other words, sample a uniformly random prime between $2^{b-1} \leq p < 2^b$). Primes are pretty dense (a random b-bit number is prime with probability $O(1/b)$), so it's enough to just pick random numbers and check if they're prime.

One simple test for primality uses Fermat's Little Theorem: if p is prime, then for any $0 < a < p$,

$$a^{p-1} \equiv 1 \pmod{p} \ .$$

It turns out that the converse is almost true: for almost all random numbers m, if $2^{m-1} \equiv 1 \pmod{m}$, then m is prime (there are some composite numbers m such that this is also true, but these are exponentially rare). This is called the Fermat Primality Test (with base 2). Implement this test, and use it to write a method to generate a random prime with b bits.

Generate a prime with 128 bits. What prime did you pick?

(c) Picking a Safe Prime and a Subgroup of Prime Order Often, we want to generate a *safe prime* p of the form $p = 2q + 1$, where q is also a prime. (This is particularly useful for discrete-log based systems, like ElGamal or Diffie-Hellman.) Write a method to generate a safe prime.

Then, write a method to pick a generator of Q_p^* (the group of quadratic residues modulo p). **Hint:** How could you verify whether a particular element of $\mathbb{Z}_p^*$ is a quadratic residue? What would it's order in $\mathbb{Z}_p^*$ be? About half of the elements are quadratic residues, so it's okay to just sample and verify.

Generate a safe prime with 128 bits and find a generator of Q_p. What prime and generator did you pick?

(d) Breaking ElGamal over $\mathbb{Z}_p^*$ Consider a variant of the ElGamal cryptosystem using $\mathbb{Z}_p^*$ for a safe prime $p = 2q + 1$: we use a generator g of $\mathbb{Z}_p^*$ (with order $p - 1$), a private key $1 \leq x \leq p - 1$, a public key g^x, and encrypts a message $m \in \mathbb{Z}_p^*$ as $(g^y, g^{xy} \cdot m)$.

Unfortunately, this scheme isn't semantically secure, because an adversary can learn whether m is a quadratic residue or not. Describe a method of determining this. **Hint:** What you get when you multiply two quadratic residues? When you multiply a quadratic residue by a quadratic non-residue? Then, use these facts to determine whether m is a quadratic residue.

Now, implement this attack: given $p = 2q + 1$, g, g^x, g^y, and $g^{xy}m$, write a function that outputs whether m is a quadratic residue.

Run your algorithm on the following input, and tell us whether m is a quadratic residue.

$$\begin{aligned} p &= 261559759947351029532457942104910865303 \\ g &= 194286524128031642142474107184510601326 \\ g^x &= 198945838169134496994751864693096545284 \\ g^y &= 162960645829528127846175960244367199327 \\ g^{xy}m &= 181937067363429702065627884694752413851 \end{aligned}$$

Problem 3-2. Computing on Secret Shares

In this problem, we will explore an interesting and useful property of Shamir's secret sharing scheme, which is that it admits additive homomorphisms over secrets.

Suppose Alice has a secret s that she has shared among n friends using the t-out-of-n Shamir's secret sharing scheme. Now, suppose she wants to update the secret shared among these friends to $s + 1$.

(a) Explain how Alice can update the secret to $s+1$ *without* knowledge of s. Your solution should involve Alice sending an update to all n parties that can be applied in parallel. Explain why your answer is correct.

Now, suppose that Alice has a secret s_A and Bob has a secret s_B. They have shared this secret among n friends using the t-out-of-n Shamir's secret sharing scheme using the same prime p. Assume Alice and Bob have the *same* set of n friends. In other words, if we call call $A_1, \ldots, A_n$ the shares of s_A and $B_1, \ldots, B_n$ the shares of s_B, then friend i has both A_i and B_i. Suppose the friends wish to compute shares of the secret $s_A + s_B \mod p$.

(b) How can the friends compute these shares without receiving any additional information? Argue the correctness of this process.

Suppose a group of n billionaires want to compute their average net worth without revealing their own net worth. We assume that there exists a private communication channel between each pair of billionaires, and we assume that the group has agreed on a prime p and a threshold t. Furthermore, we assume that all billionaires will follow the protocol you give in part (c). The only security guarantee that we require is that, after the protocol is complete, any group of $t-1$ billionaires that collude to determine the net worth of any of the other billionaires learns no additional information about another billionaires net worth beyond what can be learned from the average and from the net worth of each of the t-1 colluding billionaires.

(c) How can these billionaires use Shamir's Secret Sharing to achieve their goal? Argue that your protocol is correct and achieves the security requirement described above.

Problem 3-3. Block Cipher Attacks and MACs

Suppose Alice wants to send Bob an encrypted and authenticated message M. Alice and Bob share a random (long) secret key K. In this question, we assume the standard notion of a secure MAC (i.e., security against adaptive chosen message attacks - see Lecture 7 notes for details).

(a) For this part of the question, assume we have a standalone CPA-secure symmetric encryption scheme, and a standalone secure MAC. As stated in lecture 7, if we encrypt using this CPA-secure encryption scheme and then compute a MAC on the ciphertext (and append that to the ciphertext), we get a CCA-secure encryption scheme. In other words, we showed that for an encrypted ciphertext $C = Enc_{K_1}(M)$, that sending $C||MAC_{K_2}(C)$ was CCA-secure (provided $Enc_{K_1}(M)$ is CPA-secure and $MAC_{K_2}(M)$ is a secure MAC). However, what happens if we instead compute the MAC on the plaintext, then encrypt the plaintext and append that MAC to the resulting ciphertext, i.e. we send $Enc_{K_1}(M)||MAC_{K_2}(M)$? Give an example of an attack that breaks CCA-security for this scheme.

(b) Recall that in CMAC, we use a key K_1 for all blocks except the last, and a fresh key K_2 for the last block. Let's suppose Alice and Bob are experimenting with the security of CMAC, and are trying to develop a secure MAC scheme using CBC.

1. What can go wrong if we use the same key K for all the blocks in CMAC?
2. In their variant of CMAC, Alice and Bob decide to use key K_2 in the first block, and K_1 in all other blocks. Explain why this does not solve the problem and give an example of an attack that breaks this MAC.
3. Now Alice and Bob decide to use CMAC - all blocks use key K_1 except the last block, which uses key K_2. However, they decide not to fix the IV value at 0, (i.e. the IV can vary). Does this give a secure MAC?