MIT 6.875J/18.425J and Berkeley CS276
Foundations of Cryptography (Fall 2020)

Problem Set 2: Released September 15, Due September 29

The problem set is due on **Tuesday September 29 at 10pm ET/7pm PT**. Please make sure to upload to the Gradescope course webpage by the deadline (all registered students should have access to this webpage on Gradescope). Be sure to mark on Gradescope where each problem's solution starts. Typed solutions using LaTeX are strongly encouraged (template provided on the course webpage).

Collaboration is permitted; however, you must write up your own solutions individually and acknowledge all of your collaborators.

Problem 1. PRF Variants

In this problem, we will consider variants of a pseudo-random function, and show that they can be constructed from standard pseudo-random functions.

Consider the following variant of a pseudo-random function. We define a *summable PRF* to be a PRF family $\mathcal{F} = \{\mathcal{F}_n\}_{n\in\mathbb{N}}$ with $\mathcal{F}_n = \{F_k : \{0,1\}^n \to \{0,1\}^n\}_{k\in\{0,1\}^n}$, along with an additional PPT algorithm $S : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ such that

$$S(k,a,b) = \sum_{x=a}^{b} F_k(x).$$

In other words, F_k is a PRF such that, in addition to being able to compute $F_k(x)$ on any given point x (given the key k) in polynomial time, you can also compute sums of evaluations over any interval of inputs $x \in [a,b]$, given the key k, in polynomial time. We identify $a, b \in \{0,1\}^n$ with natural numbers between 0 and $2^n - 1$, so in particular, these sums may be exponentially long.

1.1 Show that if PRFs exist, then so do summable PRFs.

We will now consider *delegatable* PRFs. Specifically, we want a PRF family $\mathcal{F} = \{\mathcal{F}_n\}_{n\in\mathbb{N}}$ with $\mathcal{F}_n = \{F_k : \{0,1\}^n \to \{0,1\}^n\}_{k\in\{0,1\}^n}$, with an additional two PPT algorithms R and S, where $R : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ takes k as input, as well as some string $z \in \{0,1\}^m$, with $m \leq n$ and outputs a string k' such that algorithm $S : \{0,1\}^n \times \{0,1\}^{n-m} \to \{0,1\}^n$ satisfies that $S(k', y) = F_k(z + y)$, where here $z + y$ denotes the concatenation of the strings z and y.

Additionally, we require that every PPT algorithm on input k', x (with x not expressible as $z + y$ for any y) can determine $F_k(x)$ with only negligible probability.

That is, the algorithm R takes in the key, as well as some "prefix" z, and outputs another key k'. Now, using S and k' one can compute the PRF value for any x whose prefix is z (that is, any x whose first m bits are the same as z). However, even knowing k', it is not possible to compute $F_k(x)$ for any x's other than those which have z as a prefix.

1.2 Show how to use a PRG to construct a delegatable PRF.

Problem 2. A Variant of AES

One can consider the following variant of AES-256 where for every i, in the ith round, the function $F(K_i, y) = y^{-1} + K_i \bmod p$ is computed, where we define $0^{-1} = 0$. That is, our key is a sequence of integers $K = (K_1, ..., K_r)$, all modulo some large prime p (for example, p can be 256-bit).

On input x, we first compute

$$x_1 = x^{-1} + K_1 \pmod{p},$$

and then

$$x_2 = x_1^{-1} + K_2 \pmod{p},$$

and so on, until at last we compute the output $x_r = x_{r-1}^{-1} + K_r \pmod{p}$. Call this composition of all of the Fs the function $G((K_1, ..., K_r), x) = G_K(x)$.

Let's examine some properties of this function G.

2.1 Show that G is a permutation for all choices of $K_1, \ldots, K_r$.

2.2 Show that G is not pseudo-random.

Problem 3. A Subexponential Algorithm for Discrete Log

We are given inputs integers p, g, and y (where p is prime, and g is a generator of $\mathbb{Z}_p^*$), and we want to find $z_0 = \log_g(y) \pmod{p}$.

Today, the best publicly known algorithms for solving the discrete logarithm mod a general prime p take time $2^{O(n^{1/3}(\log n)^{2/3})}$, where n is the number of bits in the binary representation of p. In this problem, we will not describe that algorithm, but instead outline a different (simpler) subexponential ($2^{o(n)}$-time) algorithm for finding discrete logs mod an n-bit prime p. This approach has found applications not only in finding discrete logs mod p, but also for finding discrete logs in other fields.

We begin by defining a *smooth number*:

Definition 1 (b-smooth number)**.** *A number n is b-smooth if all of its prime factors are at most b.*

For example, $2250 = 2 \cdot 3^2 \cdot 5^3$ is 5-smooth, but it is not 4-smooth.

3.1 Say you are given an e such that $y' = (g^e)y$ modulo p viewed as an integer is b-smooth (that is, the smallest positive integer that is in the residue class of y' mod p is b-smooth). Use p, g, e, and y to find a linear equation (mod some number) relating $z_0 = \log_g(y)$ with $z_1 = \log_g(p_1), z_2 = \log_g(p_2), \ldots, z_k = \log_g(p_k)$, where $p_1 < p_2 < \cdots < p_k$ are all the primes less than or equal to b, in time polynomial in b

It turns out smooth numbers are pretty common (common enough that we can use them to find a subexponential algorithm). Specifically, we have:

Theorem 2 (Canfield-Erdos-Pomerance)**.** *Define $N(a, b)$ to be the number of b-smooth numbers less than or equal to a. If $c < \log x/(2 \log\log x)$, as c and x approach infinity, it holds that:*

$$\frac{1}{x} N(x, x^{1/c}) = c^{-c+o(c)}.$$

To find the discrete log of y, we will create a solvable system of linear equations, one of whose variables is $z_0 = \log_g(y)$. In the next subproblem we describe how to find the linear equations, and then in the final subproblem we describe the full algorithm.

3.2 Set the smoothness bound $b = 2^{\sqrt{n \log n}}$. Find a randomized $2^{o(n)}$-time algorithm for finding an e such that $g^e y \bmod p$ is b-smooth.

3.3 Describe a $2^{o(n)}$-time randomized algorithm to create $k+1$ distinct linear equations (none of which is obviously a linear combination of the others) with the following $k+1$ unknowns[1]: $z_0, z_1, \ldots, z_k$.

3.4 Describe a subexponential algorithm for solving discrete log modulo p. Analyze the run-time of your algorithm, expressed as $2^{O(f(n))}$, for some function f. Try to make it as fast as possible!

Problem 4. Factoring and Quadratics

Let $n = p_1 p_2 p_3 \cdots p_k$, where $p_1 < p_2 < p_3 < \ldots < p_k$ are prime numbers, for an arbitrary integer k.

4.1 Given a black box that can find square roots mod n, show that n can be efficiently (polynomial time in the number of bits of n, and in k) factored into its primes.

4.2 Suppose this black box only works on 1% of the inputs mod n. Show that n can still be efficiently (polynomial time in the number of bits of n, and in k) factored into its primes.

For the rest of the problem, you may assume $n = pq$ for two primes p and q.

4.3 Suppose you have a black box that on input a finds a solution to the equation $x^2 + ax + 1 = 0 \bmod n$ (if such a solution exists). Show that n can be efficiently (polynomial time in the number of bits of n, and in k) factored into its primes.

[1] For the sake of this problem, we assume (without proof) that these m linear equations with m unknowns is not underdetermined, and can be solved to find $\log_g(y)$.