The University of North Carolina at Greensboro — Handout 7
CSC 580: Cryptography and Security in Computing — March 14, 2016
Prof. Stephen R. Tate

Assignment 4 – Due Thursday, March 24

Like Assignment 3, this is a slightly shorter assignment, with only 9 days to complete it rather than a full two weeks. Because of this, it will be graded out of 50 points, and count half as much as the regular-length assignments.

1. It's clear that repeating a deterministic block cipher, as in ECB mode, is not secure. But what about repeating a secure (non-deterministic) cipher? In other words, let $E_K(P) \rightarrow C$ be an IND-CCA encryption scheme, and then define a two-block version of this by $E2_K(P_1, P_2) \rightarrow (C_1, C_2) = (E_K(P_1), E_K(P_2))$. It turns out that this construction is IND-CPA secure, but *not* IND-CCA secure. Prove the second part of that statement (in other words, give an adversary that wins the CCA game against the $E2$ two-block encryption scheme — like almost all CCA attacks, the trick is to disguise your decryption oracle requests so they don't exactly repeat the challenge ciphertext). Make sure you analyze the advantage of your adversary!

2. (*Note: Taking large modular powers is tricky, but modern programming languages have good support for this — for example, in Java you can use the* `modPow` *function in the* `BigInteger` *class, and in Python you can use the built-in* `pow` *function, where* `pow(a,x,n)` *computes* $a^x \bmod n$.) Consider the value $n = 8911$ (this is a composite number with factors 7, 19, and 67).

   (a) Select three different random a values in the range $2, \ldots, 8909$ that are relatively prime to n, and calculate $a^{n-1} \bmod n$ for each of these three a values. Does it seem that n behaves like a prime number as far as Fermat's Little Theorem is concerned?

   (b) Use your random a values from part (a) to run the Miller-Rabin primality-testing algorithm on n, showing each step. If your first a value returns "composite" you can stop with just that one simulation — otherwise, try the other a values until you get "composite."

3. What are the primitive roots of 31? (A very simple program can quickly solve this problem.)

4. All traditional public-key algorithms rely heavily on computing large modular powers, and in this problem you will explore the computational cost of modular powering. This problem assumes you will program this in Java — if you really want to use Python or another language, talk to me about this as an alternative.

   (a) Implement a function in Java that takes a single parameter, representing a number of bits, and does the following: Use the `BigInteger` constructor that generates a random big integer with the specified number of bits to generate three different random big integers, say a, b, and n. Next, simply call the `BigInteger` method `modPow` to compute $a^b \bmod n$ and return the result. Test this on some very small values (like 3 bits), print the generated values and results, and check the results using a calculator. For this part, turn in a printout of your function and show the results of your testing.

(b) Use the "profiling" tool in NetBeans to benchmark this function (you're really profiling the built-in `modPow` method on random inputs) and see how long it takes to compute modular powers on 4000, 8000, and 16000 bit inputs (you can remove the printing functions you put in for part a). You should run each input size 3–5 times and record the average time (more iterations are better — think about automating this!). If you are unsure how to use the profiling tool, documentation is available on the class web site.

(c) Modular powering algorithms generally take $\Theta(n^c)$ time, but what exactly is c? Figure it out! Here's what you need to do: Assume that the running time is $T(n) = k \cdot n^c$ for some constants k and c, and use the times you measured for 2000 and 4000 bit powering to plug in to write out formulas for $T(4000)$ and $T(8000)$. Now you have two equations with two unknowns — solve for c! Repeat the calculation using $T(8000)$ and $T(16000)$ and see if your results are consistent. You may very well see some small inconsistencies here — don't worry about this for now!

(d) Now that you know the running time of `modPow`, use this to estimate the running time for 32,000 bit inputs. Calculate this using the values you found, and *then* run a test with this input size and report on how accurate you were.

(e) Given all of these results, why might there be some inconsistencies between measurements? Are there other issues that come into play other than just CPU speed? Can you think of a way in which cache size might make a difference?