CMPUT 331, Fall 2023, Assignment 9

Before beginning work on this assignment, carefully read the Assignment Submission Specifications posted on eClass.

You will produce a total of four files for this assignment: "a9p1.py" for problem 1, "a9p3.py" for problem 3, "a9.txt" for problem 2, and a README file (in either PDF or plain text). **If your code requires any external modules or other files to run, include them in your submission as well. Your submission should be self-contained. Modules from the textbook must not be edited – if you wish to modify code from the textbook, put it in a module with a different name. Attribute any code you use, even if it is from the textbook.**

Problem 1

The security of the RSA cipher depends on there being an infinitely many prime numbers: generating large prime numbers is, as far as we know, far easier than factoring the products of these primes, meaning that, no matter how fast computers become, we will always be able to find products of larger prime numbers than potential cryptanalysts can factor. In short, the maximum practical key size is growing much faster than the maximum key size that can be hacked in a reasonable amount of time.

If there were only finitely many prime numbers, this would not be true, since computers could eventually become fast enough that any product of two prime numbers could be factored quickly. In this assignment, you will put this idea into practice by building an RSA cipher hacker which works under the assumption that there are finitely many prime numbers. This shows that the infinitude of primes, as proven by Euclid, is necessary for RSA to be a secure cipher.

Create a module called "**a9p1.py**" which contains a function "**finitePrimeHack(t, n, e)**". The "**finitePrimeHack**" function should take 3 parameters, in this order: a threshold t, and n and e, which together form a public RSA key. The function should assume that there are no prime numbers larger than t. It should then discover the values of p and q, compute the value of d (the inverse of e modulo $(p-1) \times (q-1)$) and returns a tuple containing p, q, and d in that order. For this assignment, it will always hold that $p \leq q$ (that is, the smaller of the two primes should come first; this constraint is simply to fix the ordering of primes in your solution).

For example: `a9p1.finitePrimeHack(2**16,2604135181,1451556085)` should return (48533, 53657, 60765). This example function call should take under 10 seconds to run on the lab machines.

Problem 2 Short answer

Included with the assignment are five public key files, each specifying a value of $keySize$, n and e, generated using the "makeKeyFiles" function from the textbook code. Using your code from Problem 1, for each of the five key files, determine the values of p, q, and d (that is, the two primes whose product is n, and the remaining part of the private key). Again, to fix the order of the primes, assume $p \leq q$. Note that the threshold is $2^{keySize}$. Report your results in a9.txt *in the same order in which their respective keyfiles are presented* (e.g. the second line in your text file should be the return value of your solution to "2_pubkey.txt").

Problem 3

In Problem 1, we exploited the fact that if we could restrict the number of primes to guess, then we could crack the RSA cipher in a finite amount of time. Suppose now that we lift this restriction; that is, we cannot make any assumptions about how large of primes we could have. Instead, suppose we know that the block size is set to 1. Using this restriction, we can find another way to hack the RSA cipher.

Create a module named "**a9p3.py**" with a function "**blockSizeHack(blocks, n, e)**." This function will take 3 parameters in order: a list of individual block numbers *blocks*, and n and e, which together form a public RSA key. Since we cannot guarantee a threshold t which would limit the size of p and q, we cannot use the "finitePrimeHack" function from Problem 1. Instead, exploit the fact that we know the block size is equal to 1 to decrypt the blocks into the original message. You may assume that the message was encrypted using the `publicKeyCipher.encryptMessage(msg, [n, e], 1)` function call. Your function must return the deciphered plaintext string.

As an example,

```
>>> blocks = [2361958428825, 564784031984, 693733403745, 693733403745,
        2246930915779, 1969885380643]
>>> n = 3328101456763
>>> e = 1827871
>>> blockSizeHack(blocks, n, e)
Hello.
```