601.445/600.454: Practical Cryptographic Systems October 12, 2017

Assignment 3

Instructor: Matthew Green *Due: 11:59 pm October 30, 2017*

Name: ____

The assignment must be completed individually. You are permitted to use the Internet and any printed references, but your code must be your own!

Please submit the completed assignment via Blackboard.

Problem 1: Implementing the RSA cryptosystem (40 points)

For this problem you will write a Go program that implements the textbook RSA cryptosystem at a 1024-bit key strength. Your implementation will include key generation, encryption and decryption. You must do this "from scratch", meaning that the only tools you use are the following:

1. Big integer addition, subtraction and multiplication (but *not* exponentiation, which you will code yourself.)
2. A secure random number (or bit) generation package (you can use methods that output random integers, but *not* any that find random primes).

In Go, we suggest using the `math/Big` package for big number arithmetic, and the `crypto/rand` package (specifically, the the `rand.Int()` method). For grading purposes you will hand in three separate programs.

`rsa-keygen <public key filename> <private key filename>`
  Writes a public key to the first file, writes a secret key to the second file.
`rsa-encrypt <public key filename> <plaintext in decimal>`
  Given a public key file and a plaintext in decimal, prints a ciphertext to `stdout`.
`rsa-decrypt <private key filename> <ciphertext in decimal>`
  Given a private key file and a ciphertext in decimal, prints a plaintext to `stdout`.

The public and private key files should consist of *decimal formatted* integers separated by commas and parentheses, using the following format specification. The plaintext inputs should be provided on the command line, and may consist of a single decimal formatted integer with a binary representation length of up to 800 bits.

Public key. (N, e)
Private key. (N, d, p, q)

In the notation above N denotes the public modulus, p and q are the secret factors of N. e and d represent the public and secret exponents respectively. We will provide some example key files and inputs in the course repository for you to test against. You must implement the following algorithms yourself from scratch, rather than using an existing (library) implementation:

1. Implement an algorithm for determining whether a random integer is prime (*e.g.,* Miller-Rabin).
2. Implement the Extended Euclidean Algorithm.
3. Implement the square-and-multiply algorithm for modular exponentiation ($m^e \mod N$).

You must implement all three of the above algorithms yourself, using only Bignum (multiplication/addition/setting/serialization) operations. There are existing implementations of various components of the RSA algorithm online. **You must not use them.** We will check to see if you've copied/pasted code from the Internet, and doing so will be considered a failing grade. It is fine to implement from pseudocode implementations, such as the Handbook of Applied Cryptography, so long as your code is original.

Problem 2: Implementing the Rabin cryptosystem (40 points)

For this problem you will write a Go program that implements the textbook Rabin cryptosystem at a 1024-bit key strength. Your implementation will include key generation, encryption and decryption. You must do this "from scratch", meaning that the only tools you use are the same ones you used to implement your RSA algorithm. You *may* re-use code from the previous problem.

For grading purposes you will hand in three separate programs.

`rabin-keygen <public key filename> <private key filename>`
Writes a public key to the first file, writes a secret key to the second file.
`rabin-encrypt <public key filename> <plaintext in decimal>`
Given a public key file and a plaintext in decimal, prints a ciphertext to `stdout`.
`rabin-decrypt <private key filename> <ciphertext in decimal>`
Given a private key file and a ciphertext in decimal, prints a plaintext to `stdout`.

The public and private key files should consist of *decimal formatted* integers separated by commas and parentheses, using the following format specification. The plaintext inputs should be provided on the command line, and may consist of a single decimal formatted integer with a binary representation length of up to 800 bits.

Public key. (N)
Private key. (N, p, q)

In the notation above N denotes the public modulus, p and q are the secret factors of N. We will provide some example key files and inputs in the course repository for you to test against. Recall that your implementation should include the following steps:

1. Find random primes p and q of length 512 bits each (you will have to implement the Miller-Rabin algorithm for this). **Hint:** it will simplify your implementation if you choose $p \equiv q \equiv 3 \pmod 4$, though this is optional.
2. Compute $N = pq$.

3. Implement the encryption and decryption functions.

There are implementations of various components of this algorithm online. **You must not use them.** We will check to see if you've copied/pasted code from the Internet, and doing so will be considered a failing grade. It is fine to implement from pseudocode implementations, such as the Handbook of Applied Cryptography – so long as your code is original.

Problem 3: Rabin decryption implies factoring (10 points)

One of the TAs has discovered an algorithm that "breaks" Rabin encryption! The program takes in a ciphertext c and public key N, and outputs a plaintext m such that $m^2 \equiv c \, mod \, N$. This program is invoked as follows:

`rabin-crack <public key filename> <ciphertext in decimal>`
Outputs a plaintext m in decimal to `stdout` (or nothing if the program fails).

Your job is to show that if `rabin-crack` is reliable, then you can write a second program that uses `rabin-crack` as a "subroutine" to factor an arbitrary Rabin modulus N with high probability. Your program will take in a public key N; it will invoke the `rabin-crack` program (multiple times if necessary) and parse the results; and will use this result to compute the private factors p, q. It should have the following input/output behavior:

`rabin-factor <public key filename>`
Executes the `rabin-crack` program. Outputs p, q to `stdout`.

Note: The code to our `rabin-crack` algorithm is top secret and we cannot give you a copy to test with. Therefore, for testing purposes you should write your own simulated version. The easiest way to do this is to have your test version of `rabin-crack` read the corresponding secret key from a file and just run the `rabin-decrypt` code. We will use the real version for grading, of course!

You will turn in the `rabin-factor` program as the deliverable for this part of the assignment.

Problem 4: Written Questions (10 points)

Please submit (as a text file or PDF) written answers to the following questions. Many of the answers can be found in the Handbook of Applied Cryptography.

1. Is the problem of inverting the RSA function ($m^e \mod N$) equivalent to solving the factoring problem? Explain your answer. Hint: in this meaning, an equivalence implies two implications must simultaneously hold: (1) that an efficient algorithm for factoring would allow you to invert RSA efficiently, and (2) that an algorithm for efficiently inverting RSA would allow you to factor efficiently.

2. Is the problem of inverting the Rabin function ($m^2 \mod N$) equivalent to solving the factoring problem? Use the same logic as above. Hint: consider your answer to Problem 2 (above).

3. Let's say Alice and Bob each trust Thomas to generate their encryption keypair. Because Thomas is lazy, he generates a single modulus N and then finds two pairs of values $(e_A, d_A) \neq (e_B, d_B)$ and gives (N, e_A, d_A) to Alice and (N, e_B, d_B) to Bob. What terrible things might happen in this case?

4. What happens to the security of Rabin if I'm allowed to make a *chosen ciphertext* (decryption) query to a user who is using the Rabin cryptosystem?

5. Textbook RSA (what we implemented in Problem 1) is not randomized. How should you fix this problem?