Presentación

La seguridad del criptosistema RSA va unida a la dificultad de factorizar números enteros muy grandes. Dada la clave pública de un usuario, que está formada por el módulo n y el exponente e para cifrar mensajes, se puede calcular la clave privada d, (y por lo tanto descifrar todos los mensajes) si se conocen los factores primos en los que se descompone n.

Cuando los valores de n son muy grandes (por ejemplo 300 dígitos) su factorización en números primos es una tarea muy costosa. Actualmente no se conoce ningún algoritmo capaz de hacerlo en tiempo polinómico. Actualmente el algoritmo más rápido que se conoce es el General Number Field Sieve, que lo hace en tiempo subexponencial.

Por otra parte, existen algoritmos especiales que permiten factorizar números grandes si éstos tienen ciertas propiedades.

En esta práctica trabajaremos con el criptosistema RSA. Aprenderemos como funciona el proceso de generación de claves y como se cifra y descifra un mensaje. También veremos como factorizar el módulo RSA y recuperar la clave privada a partir de la clave pública en ciertas circunstancias. Finalmente, veremos qué son los números primos robustos y por qué son importantes en el proceso de generación de claves.

Objetivos

Los objetivos de esta práctica son:

1. Entender como funciona el algopritmo RSA.
2. Entender la importancia de los números primos robustos en el proceso de generación de claves.

Descripción de la Pràctica a realizar

Esta práctica consta de tres partes. La primera parte se centra en los contenidos correspondientes a la criptografía de clave pública, concretamente en el criptosistema RSA. La segunda parte estudia cómo realizar un ataque a RSA. Finalmente, la tercera parte explica los números primos robustos, que son una parte fundamental en la seguridad de RSA.

1. Implementación del criptosistema RSA (3,5 puntos)

En esta primera parte de la práctica implementaremos el criptosistema RSA. Para hacerlo, utilizaremos la librería Sympy, que es una librería de Python que dispone de ciertas funciones matemáticas que necesitaremos, como por ejemplo nextprime, isprime, mod_inverse i gcd.

Por otra parte, podemos realizar potencias modulares usando la funciónn pow de Python.

1.1. Función que genera un par de claves RSA (1,5 puntos)

La funciónn recibirá como a argumento el tamaño de la clave.

- La variable `keylen` contendrá el número de bits que debe tener la clave a generar.
- La función retornará una lista con el par de claves generadas, en formato [[e,n], [d,n]].

Ejemplo para una clave de 32 bits: [[1658920449, 2261011201], [2103761841, 2261011201]]

1.2. Función que implementa el cifrado RSA (1 punto)

Esta función realizará un cifrado mediante el criptosistema RSA.

- La variable `pubkey` contendrá la clave pública en formato [e, n].
- La variable `message` contendrá un número entero que representará el mensaje.
- La función retornará el mensaje cifrado.

Un ejempo para una clave pública de entrada [1658920449, 2261011201] y un mensaje 1324561 sería 2193168659.

1.3. Función que implementa el descifrado RSA (1 punt)

Esta función realizará el descifrado mediante el criptosistema RSA.

- La variable `privkey` contendrá la clave privada en formato [d, n].
- La variable `ciphertext` contendrá un número entero que representará el mensaje cifrado.
- La función retornará el mensaje descifrado.

Un ejemplo para la clave privada [2103761841, 2261011201] y el mensaje cifrado 1921192979 sería 1111111.

2. Implementación de un ataque a RSA (3.5 puntos)

El algoritmo de factorización p-1 de Pollard, propuesto por John Pollard al 1974, es un algoritmo de factorización de un valor n, que suponemos producto de dos números primos $n = pq$, que explota la posibilidad de que $p-1$ o $q-1$ tengan todos sus factores pequeños.

Se parte del teorema pequeño de Fermat:

$$a^p \equiv a \pmod{p}$$

Dividimos por a en los dos lados de la equación:

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

En este punto es interesante ver que podemos elevar los dos lados de la ecuación a un exponente k cualquiera y que el lado derecho siempre valdrá 1.

$$a^{k(p-1)} \equiv 1 \pmod{p}$$

Esto es interesante puesto que $mcd(a^{k(p-1)}–1, n)$ será divisible por n.

Evidentemente, no sabemos el valor de p, pero si se diese el caso de que $p-1$ tuviese todos sus factores pequeños, podríamos encontrar un número que los contenga. Una manera sencilla de hacerlo es considerar que todos los factores de $p-1$ son más pequeños que un número B y calcular $B!$. Así, si B es suficientemente grande, $mcd(a^{B!}-1, n)$ contendrá un factor de n.

2.1. Función que implementa el algoritmo $p-1$ de Pollard (2 puntos)

- La variable `n` contendrá el número que queremos factorizar.
- La variable `B` contendrá el número entero B del algoritmo $p-1$.
- La función retornará un factor no trivial de n, o 1 si no existe.

Un ejemplo es el número entero 62615533, para el que serían resultados válidos tanto 7907 como 7919.

2.2. Función que implementa la extracción de la clave privada a partir del módulo factorizado (1,5 puntos)

Una vez factorizado el módulo disponemos de la información necesaria para generar la clave privada. Implementa la generación de la clave privada.

- La variable `p` contendrá uno de los factores del módulo.

- La variable `q` contendrá el otro factor del módulo.
- La variable `pubkey` contendrá la clave pública en formato [e, n].
- La función retornará la clave privada en formato [d, n].

El conjunto de tests asociados a este ejercicio implementa un ataque completo a RSA. A partir de una clave pública se cifra un mensaje, y a continuación, se realiza el ataque. Primero se factoriza el módulo, después se genera la clave privada y finalmente se descifra el mensaje. Fijaros en que en el último test se rompe una clave RAS de 1024 bits. Este número de bits es similar al de muchas claves RSA reales. El único motivo por el que las podemos factorizar es que se han generado sin usar números primos robustos, como veremos en el ejercicio siguiente.

3. Números robustos (3 puntos)

En el ejercicio anterior hemos visto que cuando $p-1$ o $q-1$ tienen todos sus factores pequeños, la clave se puede factorizar fácilmente usando el algoritmo $p-1$ de Pollard. Lo mismo pasa cuando $p+1$ o $q+1$ tienen todos sus factores pequeños, pues en este caso se puede factorizar usando el algoritmo $p+1$ de Williams. Es necesario pues, incorporar en el proceso de generación de claves, la verificación de que p y q son robustos, es decir, que $p-1$, $p+1$, $q-1$ y $q+1$ tienen como mínimo un factor grande.

Un número primo p es robusto cuando:

1. $p-1$ tiene un factor grande al que llamamos r.
2. $p+1$ tiene un factor grande al que llamamos s.
3. $r-1$ tiene un factor grande al que llamamos t.

Un metodo habitual para encontrar un número robusto es el propuesto por Gordon, que podéis encontrar en el Handbook of Applied Cryptography, algoritmo 4.53.

3.1. Función que implementa de nuevo el algoritmo de generación de claves RSA, esta vez usando números primos robustos (3 puntos)

- La variable `keylen` contendrá el número de bits que debe tener la clave a generar.
- La función retornará una lista con el par de claves generadas, en formato [[e,n], [d,n]].

Criterios de valoración

La puntuación de cada ejercicio se encuentra detallada en el enunciado.

Por otro lado, es necesario tener en cuenta que el código que se envíe debe contener los comentarios necesarios para facilitar su seguimiento. En caso de no incluir comentarios, la corrección de la práctica se realizará únicamente de forma automática y no se proporcionará una corrección detallada. No incluir comentarios puede ser motivo de reducción de la nota.

Formato y fecha de entrega

La fecha máxima de envío es el **27/12/2019** (a las 24 horas).

Junto con el enunciado de la práctica encontrareis el esqueleto de la misma (fichero con extensión .py). Este archivo contiene las cabeceras de las funciones que hay que implementar para resolver la práctica. Este mismo archivo es el que se debe entregar una vez se codifiquen todas las funciones.

Adicionalmente, también os proporcionaremos un fichero con tests unitarios para cada una de las funciones que hay que implementar. Podeis utilizar estos tests para comprobar que vuestra implementación gestiona correctamente los casos principales, así como para obtener más ejemplos concretos de lo que se espera que retornen las funciones (más allá de los que ya se proporcionan en este enunciado). Nótese, sin embargo, que los tests no son exhaustivos (no se prueban todas las entradas posibles de las funciones). Recordad que no se puede modificar ninguna parte del archivo de tests de la práctica.

La entrega de la práctica constará de un único fichero Python (extensión .py) donde se haya incluido la implementación.