

**INSTITUTO POLITÉCNICO NACIONAL**

**ESCUELA SUPERIOR DE CÓMPUTO**

CRYPTOGRAPHY

Sandra Diaz Santiago

**REPORTE**

**Cifrado Hibrido con Firma Digital**

Alumno:

Beltrán Orozco Isaac

**INTRODUCCION**

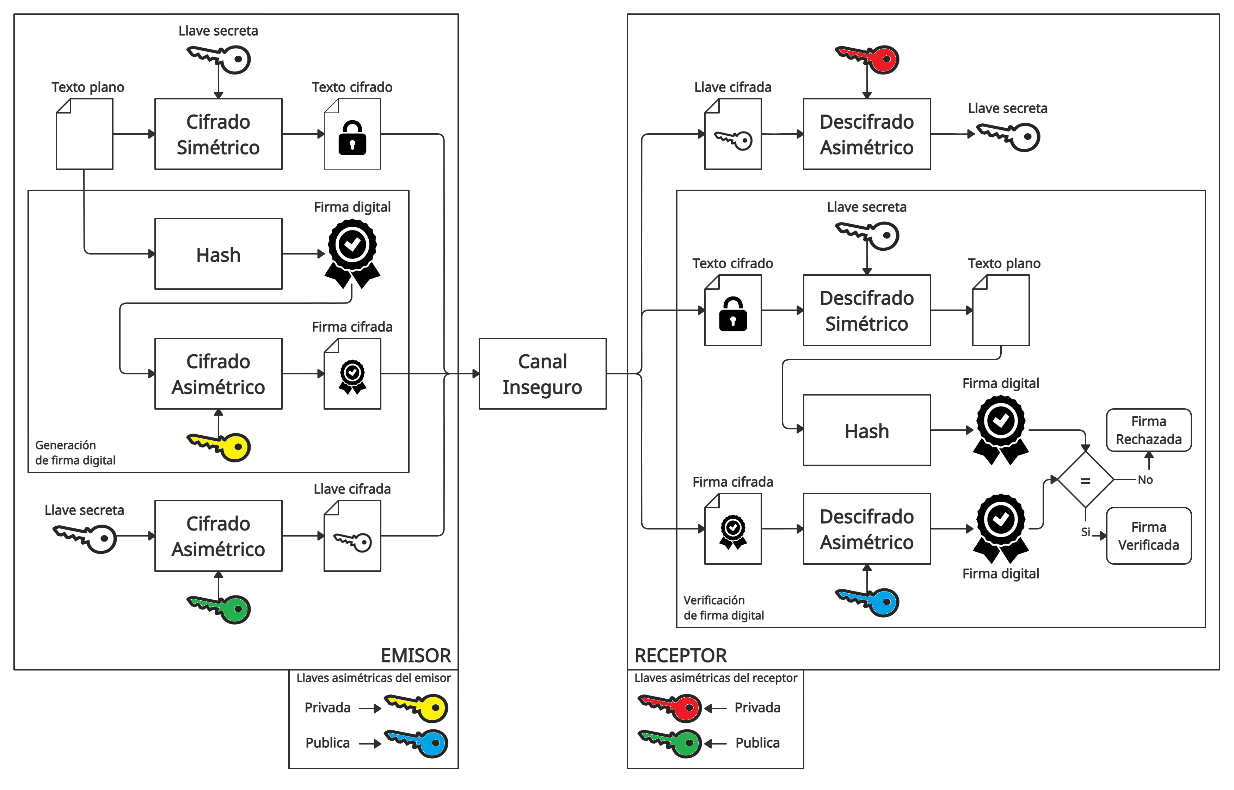
Para este proyecto, parafraseando, se nos pidió crear un programa que firmara y cifrara unos documentos (presumiblemente de texto) que el CEO de una compañía buscaba compartir de forma segura solo a personas autorizadas, de tal manera que solo dichas personas pudieran descifrar dichos documentos y verificar su firma.

También se nos pidió que este programa cumpliera forzosamente con los siguientes requerimientos:

1. Usar un algoritmo como AES o 3DES (con un modo de operación CBC, CTR, OFB o CFB), es decir un algoritmo de cifrado simétrico, también conocido como de llave de sesión o de llave secreta, cuya llave fuese generada de manera aleatoria por cada documento que cifrara con él.
2. Dicha llave debería ser, a su vez, cifrada, para poder ser compartida de forma segura entre nuestro CEO y sus destinatarios.
3. Aplicar un algoritmo de firma digital. Por ello debe ser capaz de expedir un par de llaves (pública y privada) por cada persona que las vaya a necesitar. Dichas llaves deben ser guardadas como archivos ‘.pem’. La firma debe ser guardada en base 64.
4. Exportar el texto cifrado en un archivo usando también base 64.

El sistema de cifrado propuesto para el programa que se detallará en este reporte, y que solucionaría el problema planteado, además de satisfacer todos los requerimientos exigidos, se describe en el esquema que se presenta a continuación:

**ARQUITECTURA DEL SISTEMA**

****

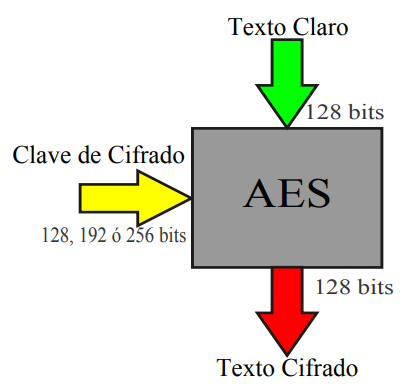
**Figura 1: Diagrama de flujo principal de nuestro programa solución.**

Ahora se procederá a explicar cada uno de los módulos que aparecen en este diagrama:

**CIFRADO SIMÉTRICO**

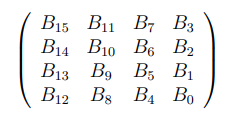
Si bien se ha colocado “Cifrado Simétrico” como nombre de este módulo, dado a que cualquier cifrado de este estilo (en teoría) resolvería el problema que se nos planteó en un inicio, para este proyecto en especifico se utilizo un cifrado AES, principalmente por qué parece ser el estándar actual en cuanto a cifrados simétricos.

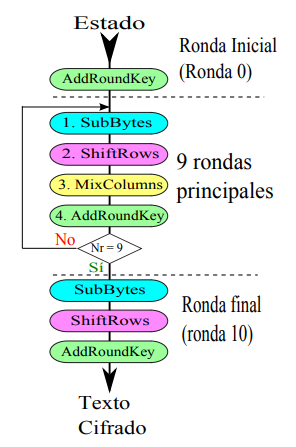
**Cifrado AES**

El cifrado AES, de la forma más resumida que me es posible explicarlo, es un algoritmo que primero divide el texto claro que hayas ingresado para cifrar en bloques de 128 bits, y cada uno de estos bloques, junto con la clave de cifrado (o llave secreta en nuestro caso), que puede ser de 128, 192 o 256 bits; serán ingresados a lo que llamare en este reporte como “Operación AES” para producir un bloque de texto cifrado de también 128 bits. Esto se repetirá hasta que todo el texto claro sea cifrado. También aclarar de una vez, que en caso de que el ultimo segmento del texto no complete el ultimo bloque de 128 bits, el algoritmo simplemente rellenara el espacio faltante.

Pero, ¿qué es lo que hace exactamente esta “Operación AES”? Pues se tratará de explicar a continuación:

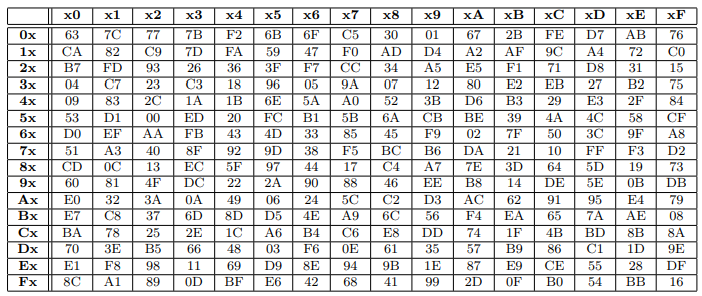
**Operación AES**

La operación AES, primero que nada, divide nuestro bloque de texto en grupos de 1 byte (igual a 8 bits) cada uno, y estos bytes los acomoda en una matriz 4x4 de arriba hacia abajo y de izquierda a derecha, como se muestra a la derecha, donde la B representa los bytes y la cadena de bits original sería: {B15, B14, B13, ,…,B1,B0}

Esta matriz, a la cual nombraremos matriz Estado, es sometida a un proceso cíclico de 10, 12 o 14 rondas, dependiendo de si nuestra llave es de 128, 192 o 256 bits respectivamente (que por cierto, esto es lo que define y da nombre a las variaciones AES-128, AES-192 y AES-256), en el cual, por cada ronda, nuestro bloque de texto pasara por otras 4 suboperaciones: SubBytes, ShiftRows, MixColums y AddRoundKey, de la manera en la que se aprecia en la imagen de la izquierda. (Imagen que, si bien muestra en específico el ejemplo para una llave de 128 bits, hay que denotar que el proceso es el mismo para las otras dos variantes de AES, donde solo la ronda final difiere un poco en cuanto a las operaciones que se le aplican a nuestra matriz).

**SubBytes**

SubBytes, o Substitución de Bytes, es una operación bastante simple que hace precisamente eso, substituir el byte que se encuentra en la celda que estemos revisando de nuestra matriz Estado por otro. Esto se puede hacer de una manera muy sencilla mediante la siguiente tabla, a veces conocida como S-box:

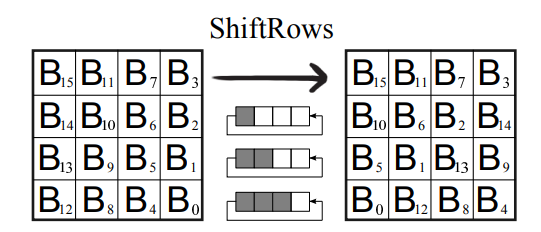
****

**Figura 2: S-Box**

Así, si el valor del byte que queremos sustituir es 1B (en hexadecimal) iríamos a la fila 1 y a la columna B de esta tabla, y en la celda en donde se encontraba 1B, después de eliminar este último, en nuestra matriz Estado ahora pondríamos un C8.

**ShiftRows**

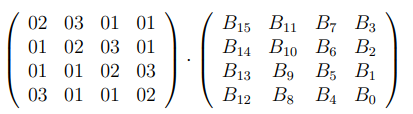
A esta matriz Estado resultante ahora se le aplicaría otra operación sencilla, ShiftRows, que simplemente rotaria sus filas de la siguiente manera:



**Figura 3: Modelo en el que se ve como es que rotan las filas de la matriz Estado. La primera fila permanece igual,** **la segunda fila se rota hacia la izquierda una posición, la tercera fila se rota hacia la izquierda dos posiciones y la cuarta fila se rota hacia la izquierda tres posiciones**

**MixColums**

La operación más compleja hasta ahora. MixColumns toma los cuatro bytes de cada columna de Estado y los interpreta como un polinomio de cuarto grado, luego multiplica el modulo x4+1 del mismo con un polinomio fijo: 3x3+x2+x+2. Es bastante más sencillo verlo como la siguiente multiplicación de matrices:



**Figura 4: Multiplicación de matrices que equivale a la operación MixColumns. A la izquierda se encuentra la matriz fija y a la derecha Estado.**

Sin embargo, para esta multiplicación hay que tener en cuenta que nuestra matriz Estado tiene que seguir estando compuesta solo por un byte por celda al final de la operación, lo cual difícilmente ocurriría si solamente multiplicáramos los elementos de estas matrices como normalmente lo haríamos. Por lo mismo hay que saber que siempre que uno de nuestros resultados supere los 8 bits hay que aplicarle un módulo x8+x4+x3+x+1 para que este se mantenga como un byte, que se traduciría a la práctica como hacer una suma xor con 100011011 a cualquier resultado de nuestras multiplicaciones entre bits que supere dicho umbral.

**AddRoundKey**

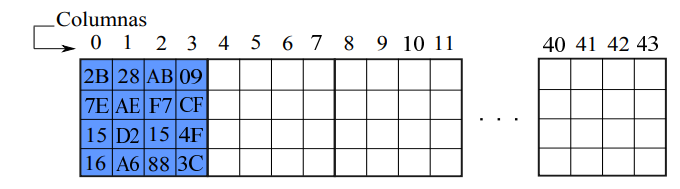
Finalmente nos encontramos con AddRoundKey, que es otra operación sencilla, al menos en parte, puesto que en si solo se trata de realizar una suma xor entre Estado y la matriz clave (si la clave es de 128 bits, si es de 192 o 256 bits solo se toma un fragmento de las mismas). Bueno, eso si nos encontramos en la primera ronda, pues en las rondas subsecuentes, con lo que tendrá que hacer la suma xor la matriz Estado será con la que llamaremos aquí como “subclave”. Pero entonces, ¿de dónde saldrán todas estas “subclaves” que necesitaremos para el resto de rondas?

**Expansión K**

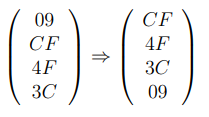
Con la expansión K es como obtenemos las distintas subclaves que usaremos a lo largo de todas las rondas del cifrado AES. Para ello lo que buscaremos hacer es obtener una matriz expandida a partir de nuestra clave original, la cual, por cierto, y para iniciar este proceso:

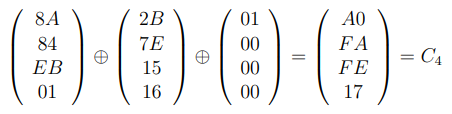
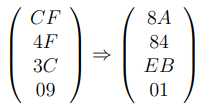
**1.-** Acomodaremos igual que nuestra matriz Estado, es decir, en una matriz 4x4, de arriba a abajo y de izquierda a derecha. Dicha matriz expandida se necesita que termine siendo de exactamente 4 filas por {4x(Nr+1)} columnas, donde “Nr” es el número de rondas que tendrá nuestro algoritmo. Así pues, notamos que AES-128 produciría 44 columnas, AES-192 produciría 52 columnas y AES-256 tendría 60, recordemos que de 4 bytes cada columna en todos los casos.

También denotar que en todos los casos las subclaves resultantes serán siempre matrices 4x4, pues no olvidemos que lo que buscamos son claves con las que poder hacer la suma xor con la matriz Estado en cada una de las operaciones AddRoundKey por las que pasara nuestro cifrado.



**Figura 5: Matriz clave expandida de AES-128 con las columnas enumeradas y solo con la clave original ingresada. En ella también podemos apreciar la división que se espera de cada una de las subclaves, es decir, que sean matrices 4x4.**

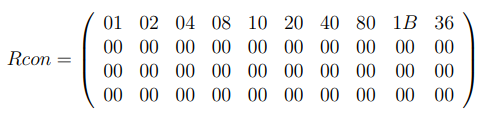
**2.-** Ahora, para iniciar la expansión, y para volver la explicación mas sencilla, nos referiremos a las columnas como Ci, donde i es el numero de la columna que estamos buscando expandir o “llenar”. Por ejemplo, la primera columna en la que debemos centrar nuestra atención es en AES-128 es en C4, pues esta es nuestra Ci=Nk, donde Nk es el número de columnas generadas por nuestra clave original (4 para AES-128, 6 para AES-192, y 8 para AES-256). Y en efecto, en este momento se encuentra vacía, para generar los bytes que la ocuparan primero debemos tomar la columna anterior a esta (Ci-1) y aplicarle una rotación hacia arriba.

**3.-** Luego a esta columna rotada se le aplica SubBytes, operación que ya explicamos anteriormente.

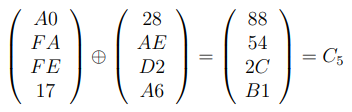
**Figura 6: Aplicación de SubBytes.**

**Figura 7: Aplicación de las sumas xor entre el resultado de la aplicación de SubBytes, Ci-Nk y Rcon(i/Nk), así como la C4 resultante.**

**4.-** El siguiente paso sería sumarle a nuestra columna resultante (con xor, por supuesto) tanto Ci-Nk (C1 en este caso) como Rcon(i/Nk) {Rcon(4/4=1) en este caso}, siendo Rcon la siguiente matriz fija (y cuyas columnas estan enumeradas del 1 al 10 de izquierda a derecha:



**5.-** Es muy importante resaltar que todo el proceso desde 1.- hasta 4.- solo se realiza con las columnas cuya i=Nk\*n, siendo n cualquier número real; para el resto de las columnas, por ejemplo C5, C6 o C7; basta con hacer la suma xor entre Ci-1 y Ci-Nk para saber cuál será su contenido:



Cuando se llegue, por ejemplo, a i=8=Nk\*n, con n=2; se repetirá el proceso que iba desde 1.- hasta 4.- para luego con C9, C10 o C11 volver a hacer solo 5.- y así continuaremos el ciclo hasta que consigamos todas las columnas requeridas por nuestra variación de AES.

Ya como ultima aclaración para expansión K, decir que el proceso aquí explicado es idéntico tanto para AES-128 como para AES-192, pero para AES-256 se agrega un paso adicional que solo aplica en cada i= (Nk\*n)+4. En estos casos, en lugar de cualquiera de los pasos anteriormente descritos, se procederá a:

**6.-** Aplicar SubBytes a Ci-1 y se sumara el resultado (con xor) a Ci-Nk (donde ya sabemos que Nk=8).

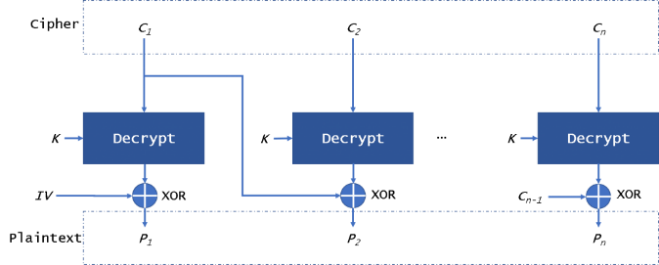
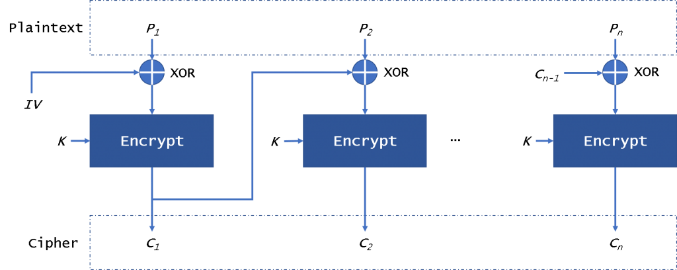
Por lo que si queremos hallar C12=(Nk\*n)+4, con n=1; simplemente se aplicara SubBytes a C11 y el resultado se sumara con C4.

**Modos (y CBC)**

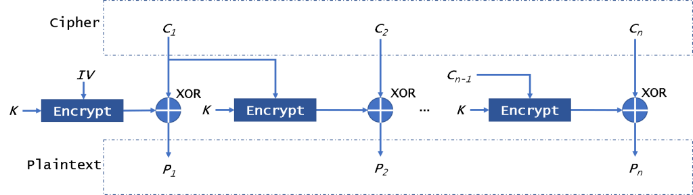
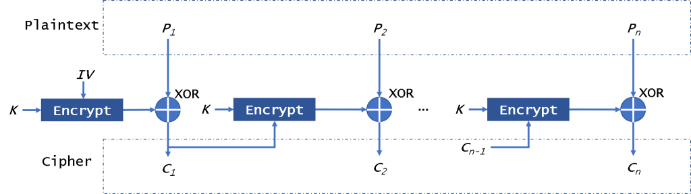
Como último punto a tratar sobre el cifrado AES profundizaremos únicamente del modo que estamos usando en este proyecto: CBC, ya que AES posee muchos otros modos que tomaría todavía más tiempo detallar, además de que también está el hecho de que una vez se explica uno se puede intuir en buena medida como funcionan los demás.

Sin embargo, para poder empezar dicha profundización forzosamente tendremos que pasar por el concepto de modo y sus generalidades, que creo podría resumir en la definición: “Son los modelos a partir de los cuales el algoritmo construye el texto cifrado”.

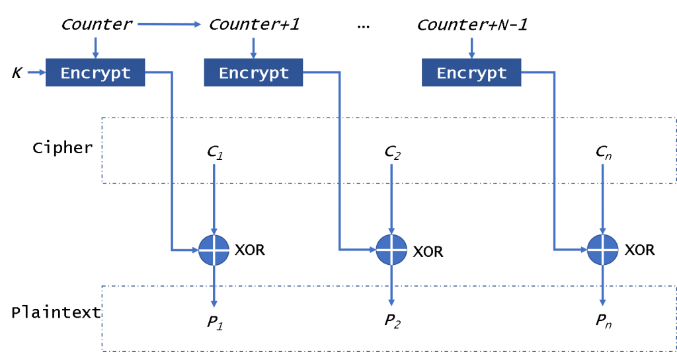
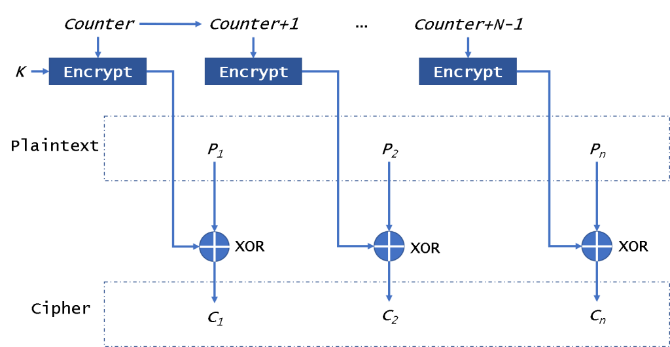
Considero que es más sencillo de entender viendo algunos esquemas de los mismos:



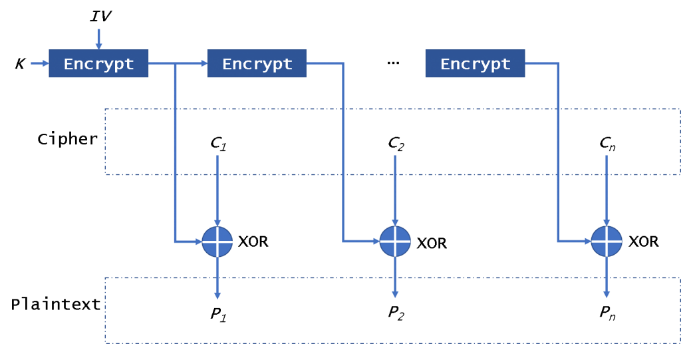
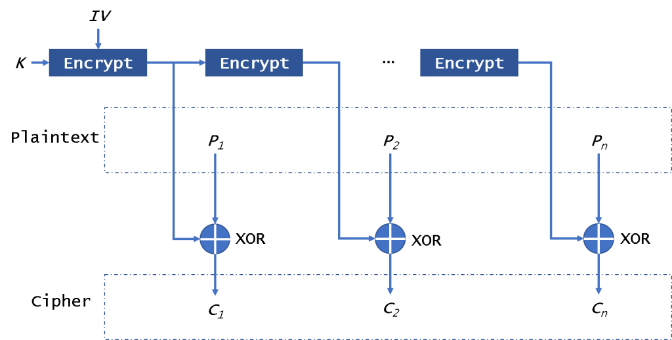
**Figura 8: Cifrado (izquierda) y descifrado (derecha) AES en modo CBC** **(Cipher Block Chaining)**

****

**Figura 9: Cifrado (izquierda) y descifrado (derecha) AES en modo CFB** **(Cipher FeedBack)**

****

**Figura 10:** **Cifrado (izquierda) y descifrado (derecha) AES en modo CTR (Counter)**

****

**Figura 11:** **Cifrado (izquierda) y descifrado (derecha) AES en modo OFB (Output FeedBack)**

Se pueden puntualizar varias cosas con los modelos ya mostrados, para dejar más claros algunos de los conceptos que se presentan:

Podemos ver como en todos se nos presenta primero, en el proceso de cifrado (a la izquierda), a Plaintext (el texto plano o claro), del cual se van tomando pequeñas partes (Pn), las cuales son los bloques de 128 bits en los que ya habíamos dicho que se dividía nuestro texto.

Vemos como estos conectan ya sea con “Encrypt” (encriptación o cifrado, que aqui, obviamente, siempre se refiere al cifrado AES) o con una suma xor, cuyo resultado produce una parte de Cn, es decir, un bloque de 128 bits del texto cifrado (o “Cipher” en los esquemas), como ya habíamos explicado antes.

También desde aquí podemos notar la presencia de nuevos conceptos, como “IV”, que en muy resumidas cuentas es un numero aleatorio que nos ayuda con los cifrados; o “Counter” en el caso de CTR, que como su nombre lo dice, se trata de un contador.

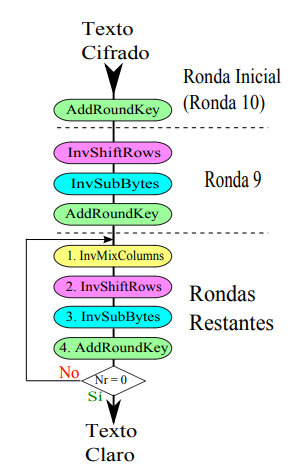
Pero a la derecha, en el lado de los descifrados, nos encontramos con el curioso fenómeno de que solo CBC parece ser el único que de hecho requiere hacer el proceso de descifrado AES, lo que es entendible por la propiedad conmutativa de la operación xor, y es que vean aquí la principal razón por la cual se optó por CBC como el modo para este proyecto, pues me parece que es el único modo, entre los propuestos como ejemplo en el planteamiento de nuestro problema, que de hecho cifra, aunque sea indirectamente, el contenido del texto plano y no solo le pasa un xor antes de pegarlo en el texto cifrado, el cual me da mucha menos confianza. Aunque claramente esta la excepción de CFB, que de hecho solo lo hace con el primer bloque de texto que procesa. Pero aún me sigue pareciendo más confiable no dejar rastro posible del texto plano en el texto cifrado.

Este elemento me parece suficientemente fuerte como para perdonar las falencias de CBC por el mismo. Falencias como el hecho de que los bloques se vuelven dependientes de los bloques que les preceden (y por tanto si falla uno fallan todos los posteriores), o que si diseño no permita plantea un cifrado rápido mediante computación paralela.

Por lo que respecta al funcionamiento del modo escogido… Considero que la Figura 8 es bastante es bastante autoexplicativa.

**DESCIFRADO SIMÉTRICO**

**Descifrado AES**

Ya habiendo detallado el módulo más grande del proyecto, nos adelantamos un poco para describir de una el módulo contrario a este, el cual será muchísimo mas sencillo de explicar con su contraparte ya desmenuzada, pues en resumidas cuentas se podría explicar cómo: aplicar la inversa de todas las operaciones aplicadas (menos de AddRoundKey, la cual se mantiene y aplica de la misma forma que en el cifrado AES).

**InvShiftRows**

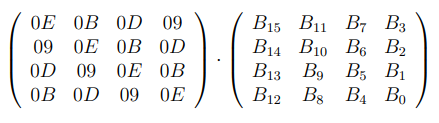
Las filas rotan el mismo número de espacios que lo hicieron antes pero ahora en el sentido contrario. La primera fila se mantiene, la segunda rota uno a la derecha, la segunda rota dos a la derecha, y la tercera tres a la derecha.

**InvSubBytes**

Se saca la matriz inversa de S-box y con ella se sustituyen los bytes de Estado de la misma manera que se hizo en el cifrado AES.

**InvMixColums**

Se trata como un polinomio cada columna de Estado y luego se multiplica el módulo x4+1 con un polinomio fijo: 9x2. Una vez mas es mucho mas intuitivo verlo como una multiplicación entre dos matrices:



**Figura 12: Multiplicación de matrices que equivale a la operación InvMixColumns. A la izquierda se encuentra la matriz fija y a la derecha Estado.**

**CIFRADO ASIMÉTRICO**

Habiendo acabado por fin con el cifrado AES, por lejos el tema más complejo de profundizar, pasamos a explicar el segundo algoritmo importante de nuestro programa propuesto: RSA.

De nuevo, el modulo se plantea como “Cifrado Asimétrico” porque, en teoría, cualquier algoritmo de este estilo puede cumplir esta función, pero de nuevo, con RSA nos apegamos a los estándares de seguridad actuales.

**Cifrado RSA**

La seguridad del cifrado RSA se basa en el hecho de que es muy sencillo multiplicar dos números primos muy grandes, pero no así hacer el proceso inverso: dado el producto de dos números primos grandes, encontrar los factores que lo produjeron. Y esto es lo único que necesita para generar un cifrado bastante poderoso:

1. Se eligen aleatoriamente dos números primos grandes ‘p’ y ‘q’ de manera aleatoria, para después sacar su producto: n=p\*q.
2. Se calcula la función de Euler del módulo n, que para dos números primos es:  
   (n)=(p-1)(q-1).
3. Se escoge un número e, en el intervalo 1 < e < (n), que sea coprimo o primo relativo con (n), es decir, de forma que el máximo común divisor de e y (n) sea 1  
   (mcd(e,(n))=1). La clave pública será (e, n).
4. Se calcula, mediante el algoritmo de Euclides extendido, el inverso multiplicativo (d) de e módulo (n), es decir, que satisfaga: d\*e 1 mod((n)). La clave privada será (d, n).

El cifrado del mensaje (m) lo realiza el emisor con la clave pública del receptor mediante la siguiente expresión: c = me mod (n) y…

**DESCIFRADO ASIMÉTRICO**

**Descifrado RSA**

el receptor lo descifra con su clave privada mediante la expresión: m = cd mod (n).

**GENERACIÓN DE LA FIRMA DIGITAL**

Finalmente, toca explicar el ultimo de los algoritmos que conformaran nuestro programa solución, la firma digital. Y será bastante sencillo de explicarla al ya haber hablado del…

**Cifrado RSA**

Puesto que se usará este mismo, prácticamente sin ninguna modificación, aunque intercambiando la llave pública y la privada en su función, y ahora usaremos para cifrar el producto de la operación…

**Hash**

Las funciones hash, en resumidas cuentas, son operaciones matemáticas que muestran las siguientes características:

1. Es libre de colisiones, es decir, para cada entrada solo hay una única salida. No debe haber nunca dos entradas que produzcan la misma salida.
2. Pueden ocultarse, es muy difícil encontrar el valor de entrada a partir de la salida que produjo la función hash.
3. Transforman sus entradas en cadenas de bits de tamaño constante (independientemente del tamaño de la entrada), conocidas como “valor hash”.

Entre algunas más, pero he aquí los principales.

Para nuestro programa solución se implemento un hash del tipo SHA-256, por estándares más que nada de nuevo, aunque esta vez por estándares educativos parece ser.

**VERIFICACIÓN DE LA FIRMA DIGITAL**

Y finalmente terminamos con la verificación de la firma digital. Aunque, si son observadores, podrán notar que ya se han explicado todos los módulos que aquí se encuentran. Como mucho recordar que el descifrado de la firma digital se hará con la llave publica en lugar de con la privada. Pero esto también es bastante intuitivo para este punto, al igual que lo poco que ha quedado sin profundizar de la Figura 1.

**FUNCIONES IMPLEMENTADAS**

A continuación, se mostrarán y explicaran las distintas funciones que se implementaron en el programa solución propuesto, las entradas de estas, su comportamiento y sus salidas.

**file\_admin.py**

El primer archivo que se revisara, file\_admin.py, contiene las funciones más básicas de nuestra aplicación, puesto que buena parte de las demás funciones hacen uso de estas. También denotar que para ello hemos hecho uso, en una pequeña medida, de la biblioteca tkinter, intentando aportar algo de estética y comodidad al programa.



{Línea 6-10}- open\_file(): Función sin entradas que utiliza tkinter casi en su totalidad para obtener el dirección de cualquier archivo que seleccione el usuario en la interfaz de tkinter. Ya con ella procede a llamar a…

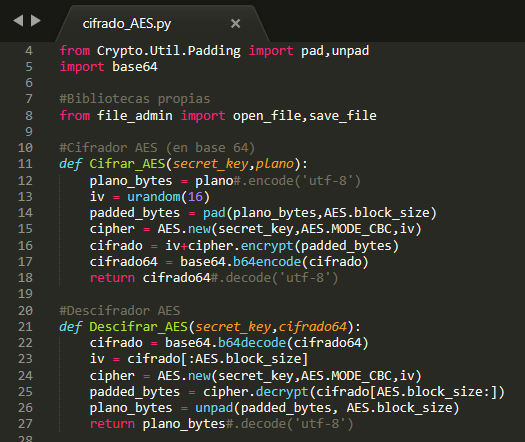
{Línea 12-15}- get\_content(): Función que recibe la dirección de un archivo, lee los bytes de este mismo y estos últimos son los que regresa.

{Línea 17-21}- save\_file(): Función que recibe el nombre de un archivo (str), el contenido del mismo (bytes) y una extensión para el archivo (str). Abre o crea el archivo que está en la dirección que le devuelve save\_path() y escribe en el los bytes que se le paso en “contenido”.

{Línea 12-15}- save\_path(): Función que recibe un nombre y extensión, ambos string, y mediante tkinter pide al usuario, mediante su interfaz, una dirección en la cual el usuario busque guardar el archivo. También se puede modificar el nombre, ubicación o extensión del archivo mediante esta. Ya con la dirección procede regresarla como respuesta.

**cifrado\_AES.py**

Empezamos con los algoritmos criptograficos. Afortunadamente en la actualidad, gracias a la librería PyCryptodome de Python, implementar este algoritmo es muchísimo mas sencillo que explicarlo:

{Línea 10-18}- Cifrar\_AES(): función que recibe una llave secreta, la cual debe ser una serie de 16, 24 o 32 bytes, muy preferentemente generados con un algoritmo de randomizacion criptográficamente seguro, como urandom(); y una cadena de bytes, idealmente correspondientes al texto plano que buscamos cifrar. Seguidamente, línea 13, generara una serie de 16 bits aleatorios, llamados “iv”, que si leyeron la sección de “Modos (y CBC)” ya deberían intuir para que nos servirá. Denotar que forzosamente debe medir 16 bytes esta cadena (para poder armar una matriz 4x4 con la que sea compatible realizar la suma xor correspondiente). Línea 14, se “paddea”, o “rellena” la secuencia de bytes que representa nuestro texto plano para que esta sea un múltiplo de 128 bits (AES.block\_size) y se pueda realizar el cifrado sin problema. Línea 15, se crea el objeto AES, al cual llamaremos “cifrador”, ya que es este quien guardara y “podrá usar” la llave secreta, el modo del AES que seleccionamos (que aquí es donde nos podemos fijar que CBC) y el vector “iv”, que precisamente nuestro modo nos exige para realizar el cifrado correctamente; para cifrar nuestra texto en la siguiente línea, la 16, en donde se “le pide” al cifrador encriptar los bytes “padeados” de nuestro texto plano. Este procederá a hacerlo de acuerdo a los atributos (es decir llave, modo, etc.) que le pasamos, lo que resultara en una cadena de bytes ya cifrada a la que, antes de guardarla, le añadiremos nuestro vector “iv” al principio, pues este es necesario para el descifrado del texto y requerimos pasárselo de alguna manera al descifrador. En la línea 17 solo codificamos la cadena de bytes resultante a base 64 con una simple función, y esto ultimo es lo que regresa nuestra función en la línea 18.

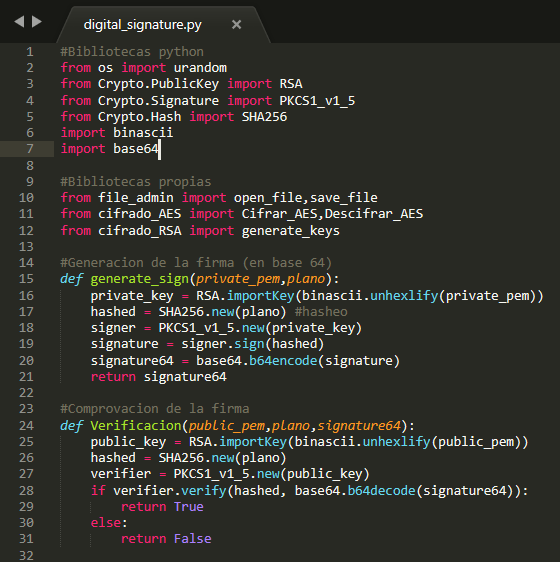
{Línea 21-27}- Descifrar\_AES(): Al igual que en la explicación teórica, esta función hace exactamente los pasos inversos de la función Cifrar\_AES(): Se recibe la llave secreta y el texto en base 64. Se descodifica de esta base y se obtienen los bytes cifrados (línea 22). Se extrae “iv” de la cadena recibida (línea 23). La cadena se vuelve a “padear” para que pueda volver a pasar por el algoritmo AES. Creamos un nuevo cifrador pero con exactamente los mismos atributos que los de la función de cifrado y pasamos a descifrar los bytes “padeados” (línea 24-25). Finalmente terminamos con el texto descifrado, aunque con unos bytes de mas que quitaremos con unpad() y son los bytes que representan nuestro texto plano lo que regresa esta función (línea 26-27).

**cifrado\_RSA.py**

Continuamos con el siguiente algoritmo, RSA, es un poco mas complejo de implementar que el anterior.

{Línea 11-23}- generate\_keys(): Función sin entradas que mediante un tamaño de llave y un numero aleatorio seguro genera un llave, que viene a ser nuestra llave privada (línea 14). A partir de esta se genera una llave publica y se busca exportar ambas como archivos .pem (líneas 15, 17, 19 y 20). Finalmente se exporta la llave publica generada como un archivo .pem (línea 22), y la misma función regresa lo que sería la llave privada.

{Línea 26-30}- Cifrar\_RSA(): Recibe una llave publica en formato .pem, le quitamos dicho formato (línea 27). Creamos un cifrador, esta vez del tipo PKCS1 (el estándar que define muchas de las propiedades de RSA) (línea 28), y ya con él procedemos a encriptar plano, que también recibimos por cierto (en forma de bytes, como siempre), y que va a ser lo que regresemos (líneas 29-30).

{Línea 32-37}- Descifrar\_RSA(): Recibe una llave privada en formato .pem y un texto cifrado. Realiza exactamente los mismo pasos de Cifrar\_RSA(), solo que el cifrador en lugar de llamar a encrypt() llama a decrypt() (linea 36), y el obvio intercambio de la llave.

**digital\_signature.py**

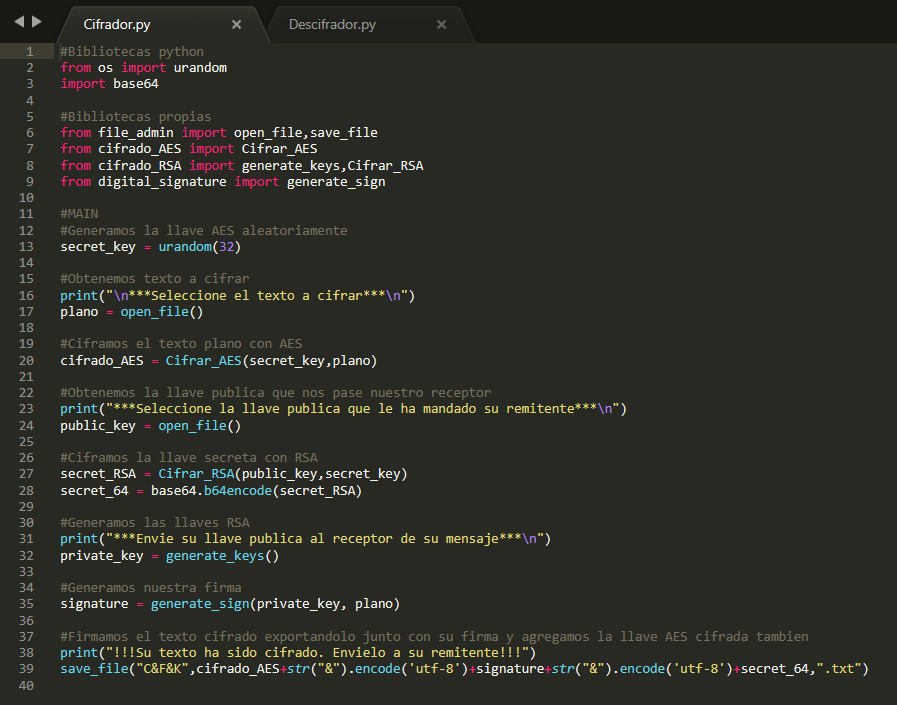
Este fue el código mas complejo de implementar, ya que básicamente aplica todo lo que se vio hasta ahora:

{Línea 15-21}- generate\_sign(): Función que recibe una llave privada en formato .pem y un texto plano. A la llave se le quita el formato .pem (linea 16). El texto plano se hashea (linea 17). Se crea un cifrador con la llave privada (linea 18), se le hace firmar el texto hasheado (linea 19) y la firma resultante se pasa a base 64.

{Línea 24-31}- Verificacion(): Recibe una llave publica en formato .pem, a la cual procede a quitarle dicho formato (linea 25); un texto plano el cual procede a hasear (linea 26), y tras crea un “verificador” con la llave publica, compara el texto que acaba de hashear con la firma en base 64 que recibió, y que en este momento procede a decodificar; para ver si es que las dor firmas son iguales. Regresa “true” y es así y “false” si no.

**Cifrador.py**

Finalmente van los archivos principales. Pues ahora que ya hemos dicho como trabajan todas las funciones, explicar estos será bastante más sencillo:



{Línea 13}- Generamos la llave secreta.

{Línea 17}- Abrimos el texto a cifrar: “plano”.

{Línea 20}- Aplicamos el cifrado AES (con la llave secreta) a plano.

{Línea 24}- Abrimos la llave publica que nos pasó nuestro receptor.

{Línea 27-28}- Aplicamos el cifrado RSA (con la llave publica del receptor) a la llave secreta, y el resultado lo pasamos a base 64.

{Línea 32}- Generamos nuestro par de llaves.

{Línea 35}- Generamos la firma para plano, la firma cifrada (con nuestra llave privada).

{Línea 39}- Guardamos tanto el texto cifrado, como la firma, como la llave cifrada, en el mismo documento: C&F&K.txt.

**Descifrador.py**

{Línea 13}- Generamos nuestro par de llaves.

{Línea 17}- Abrimos el texto a descifrar: “CyFyK”.

{Línea 20-22}- Separamos CyFyK en los 3 elementos que lo componen: el texto cifrado en AES, la firma cifrada y la llave secreta cifrada con RSA.

{Línea 25-26}- Desciframos la llave secreta (con nuestra llave privada).

{Línea 30}- Abrimos la llave publica que nos pasó nuestro emisor.

{Línea 33}- Desciframos el texto encriptado con AES (con la llave secreta descifrada)

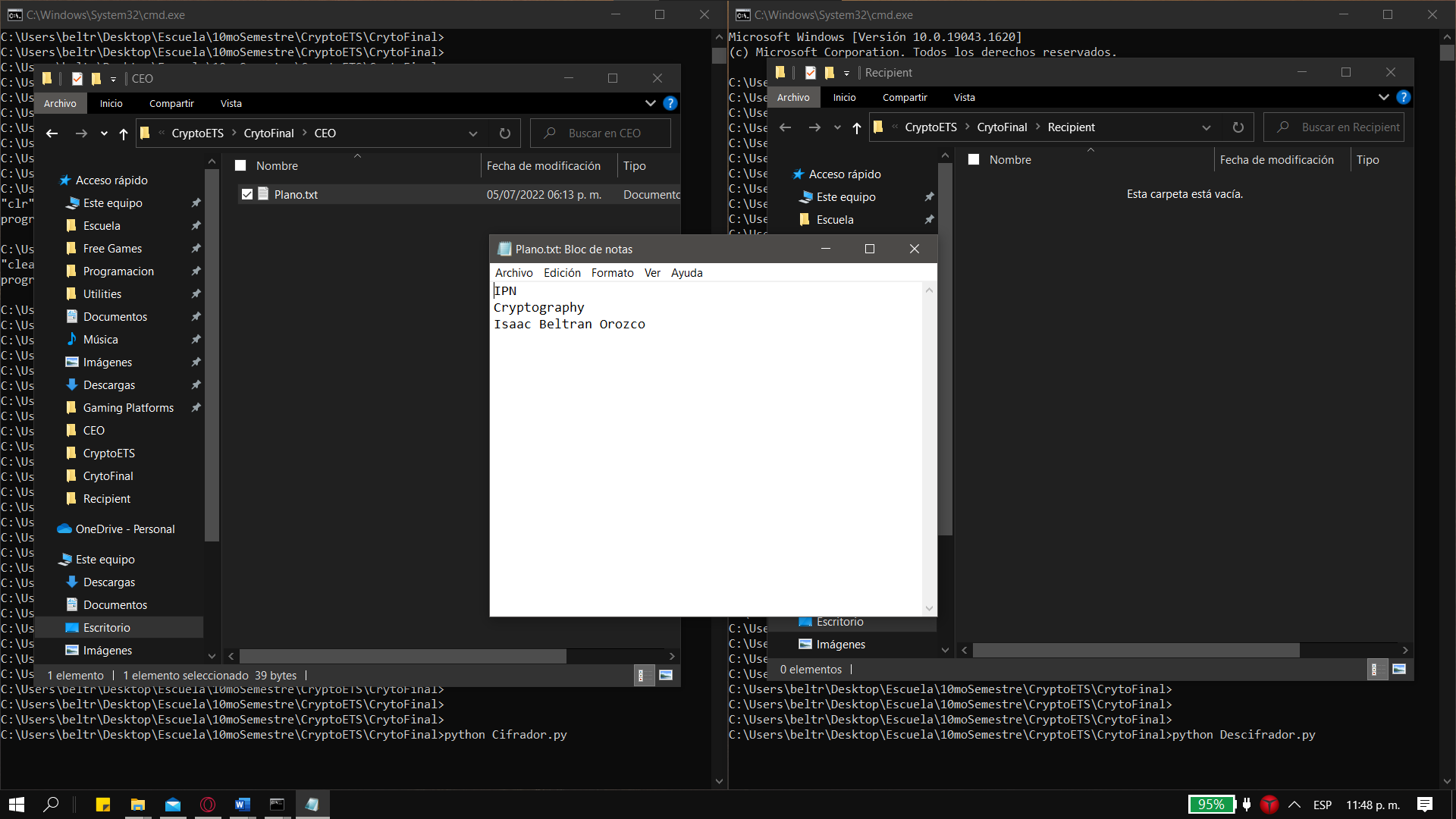
{Línea 37-40}- Verificamos que la firma incluida en CyFyK sea auntentica.

{Línea 43}- Exportamos el texto descifrado.

Texto

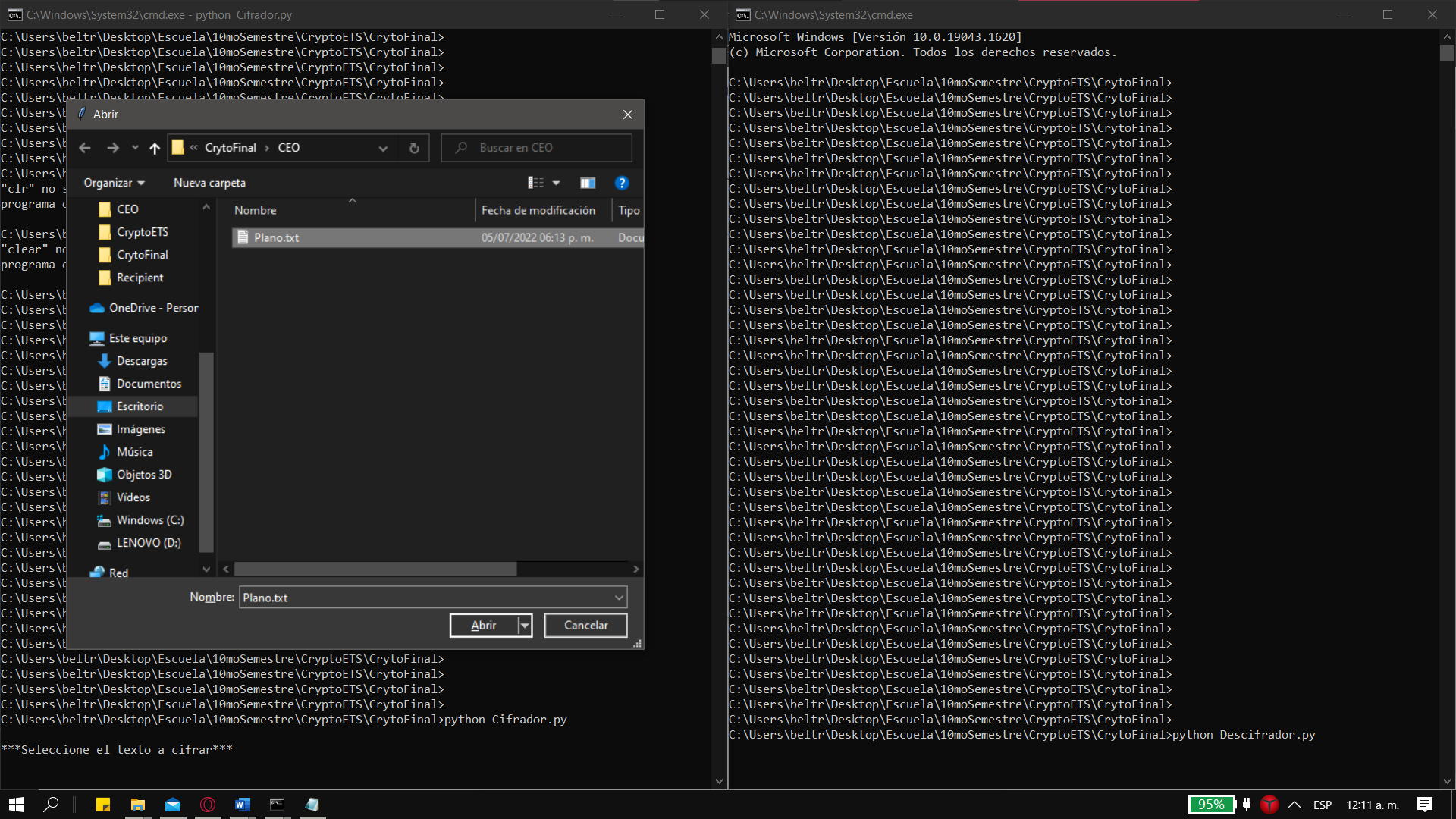
Descripción generada automáticamente

**FUNCIONAMIENTO**

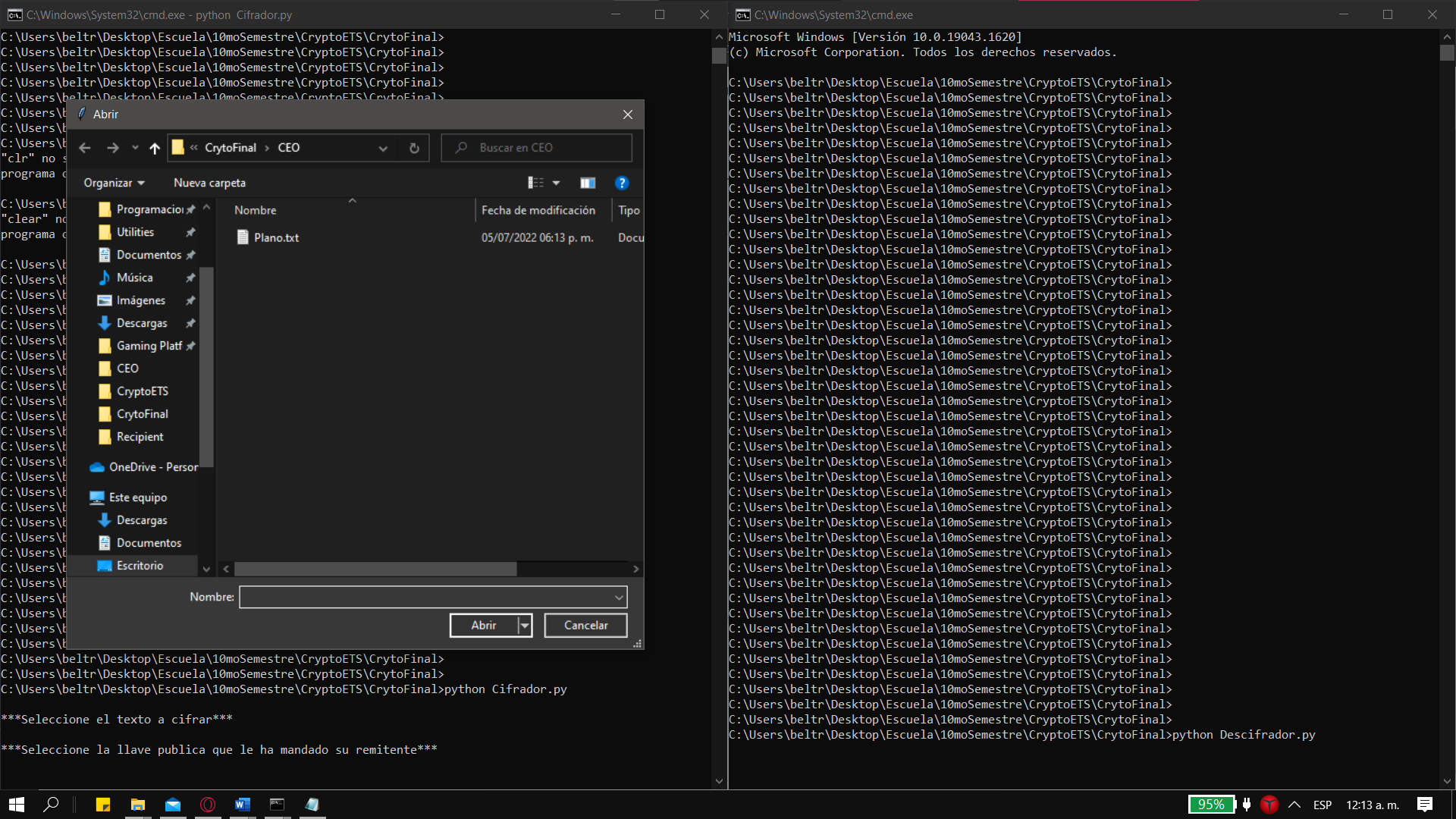
****

Nos encontramos en el estado inicial de nuestro sistema. A la izquierda se encuentra el lado que representará el escritorio de nuestro emisor, atrás esta la terminal que correrá Cifrador.py y enfrente esta abierta una carpeta llamada “CEO” que únicamente contiene un .txt llamado “Plano”, cuyo contenido se puede ver en el centro de la pantalla. A la derecha estará el espacio del receptor. Atrás, en su terminal, se ve que ahí se ejecutara Descifrador.py, y enfrente, en la carpeta del mismo, llamada “Recipient”, se ve que no hay nada. Nuestro objetivo es que el receptor reciba en esa carpeta un texto cifrado y que al descifrarlo este sea igual que el que aparece ahora en pantalla.

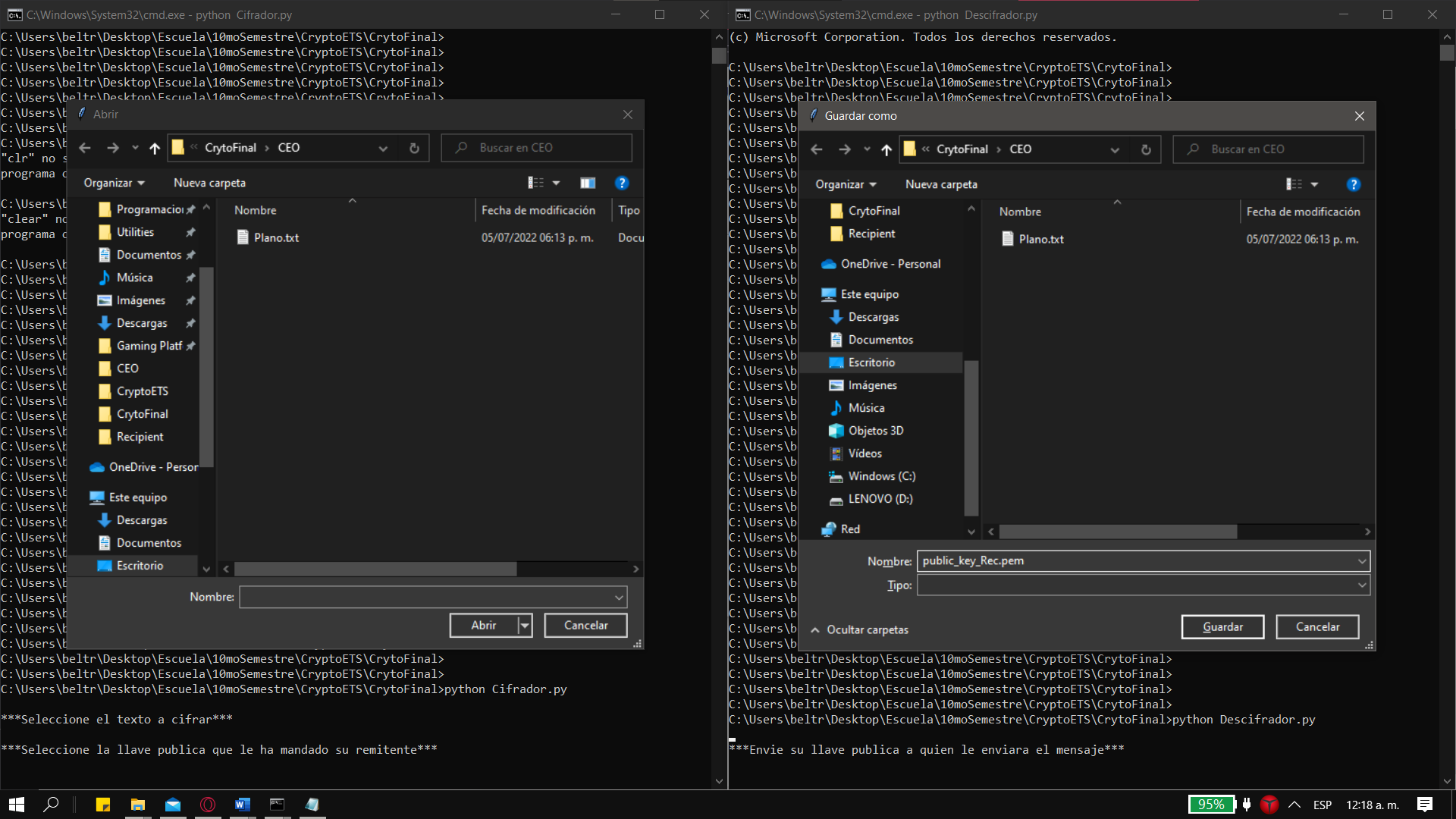
Empezaremos ejecutando Crifrador.py



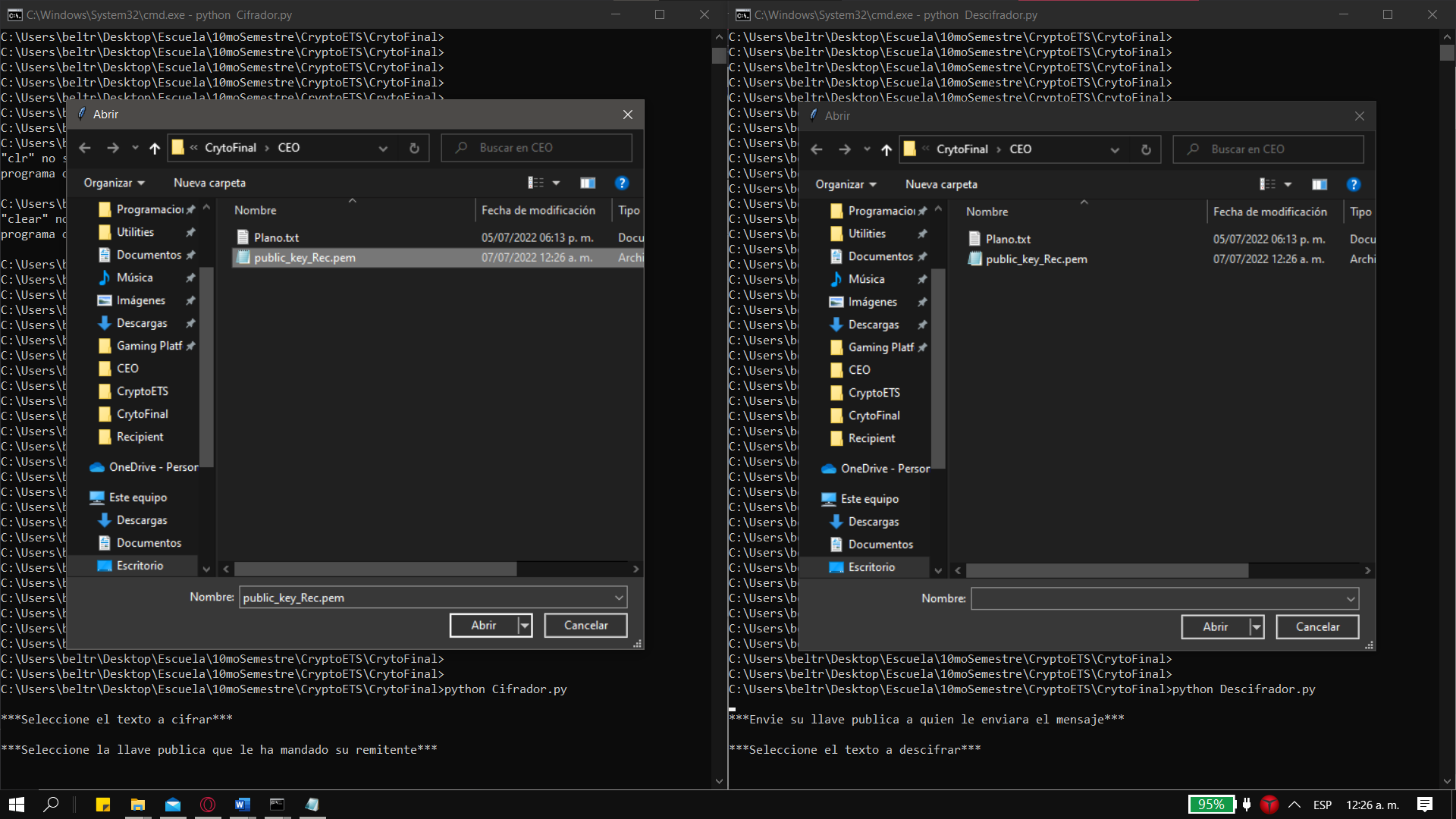
Como resultado la consola nos mostró el mensaje “Seleccione el texto a cifrar”, para luego hacer emerger una ventana que nos pide abrir un documento. Ya seleccionamos Plano.txt y procederemos a darle en abrir.



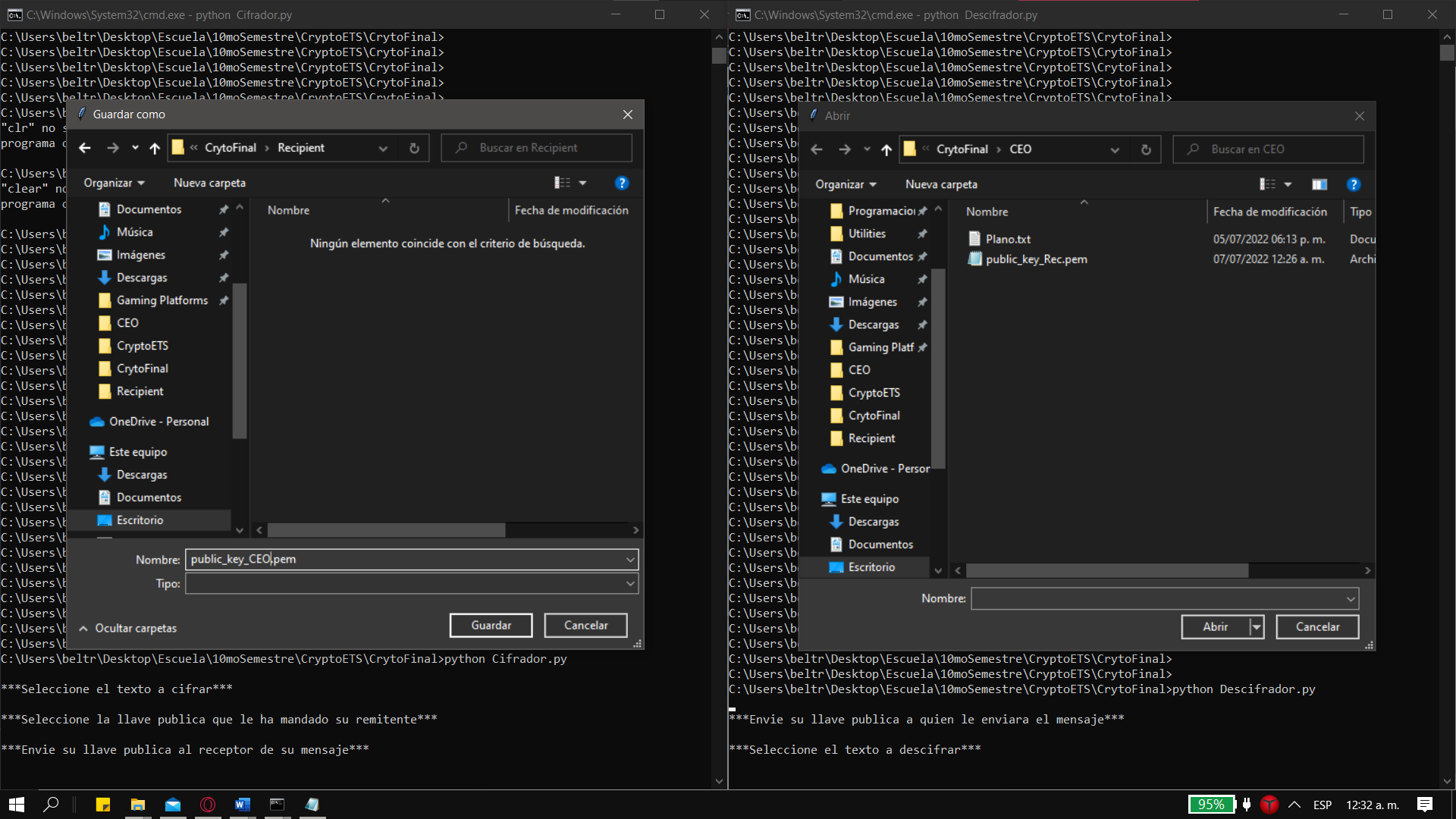
La ventana anterior se cerró y se abrió una idéntica en su lugar, pero ahora apareció el mensaje “Seleccione la llave publica que le ha mandado su remitente”. En el remitente, ya que el receptor todavía no tiene la llave que se le está pidiendo, procederemos a ejecutar Descifrador.py.



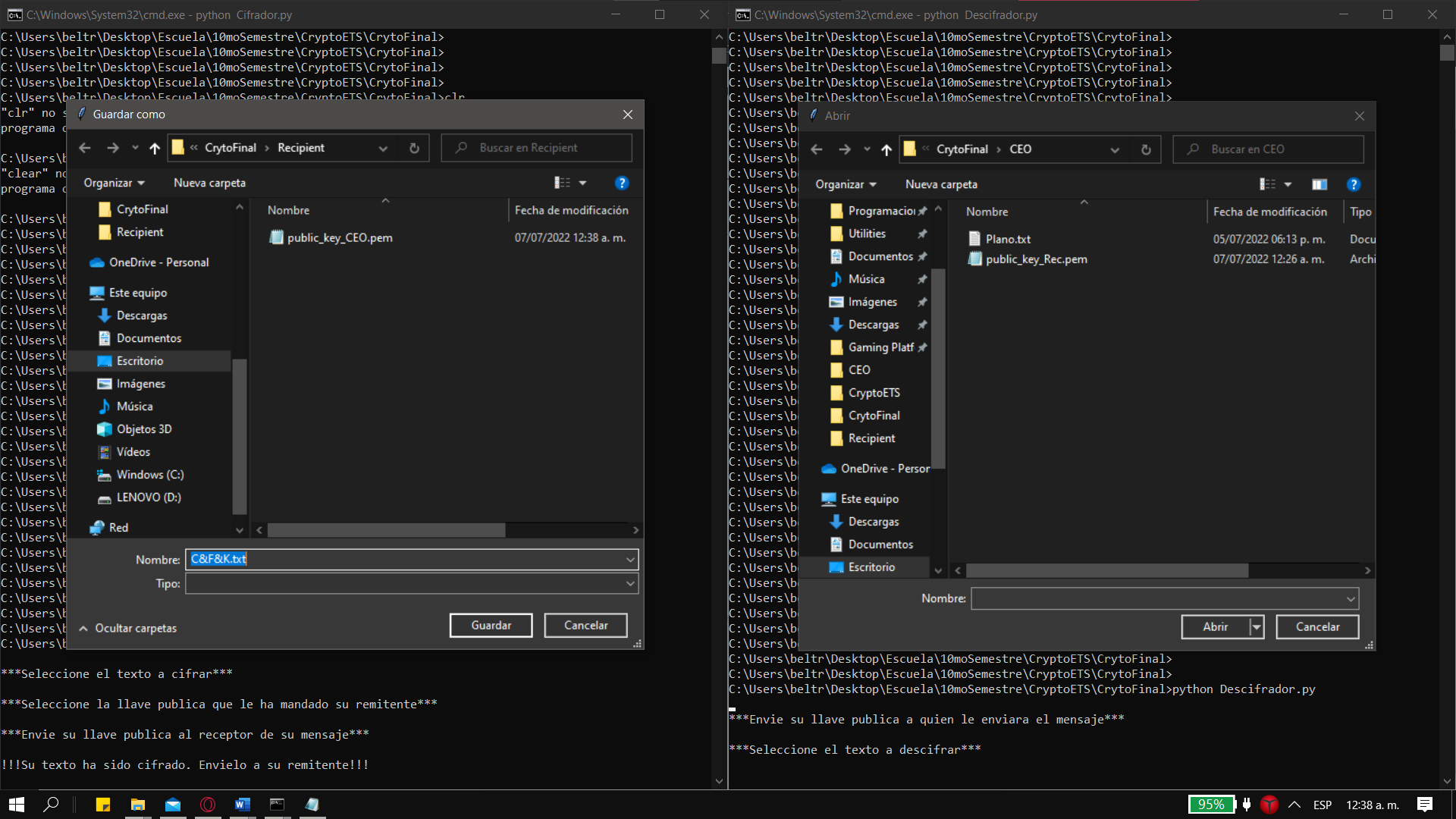
Ahora en la consola del receptor nos apareció el mensaje “Envié su llave publica a quien le enviara el mensaje” y apareció una ventana parecida a las otras, pero esta parece ser para guardar archivos en lugar de para abrirlos. El nombre predefinido del archivo que se estaba intentando guardar era “public\_key.pem”, pero lo cambiaremos a “public\_key\_Rec.pem” para que se vea que esa ha sido la llave publica que ha generado el receptor. Además, guardaremos esta llave en la carpeta del emisor, en “CEO”, para simular que le hemos pasado la llave del receptor al emisor.



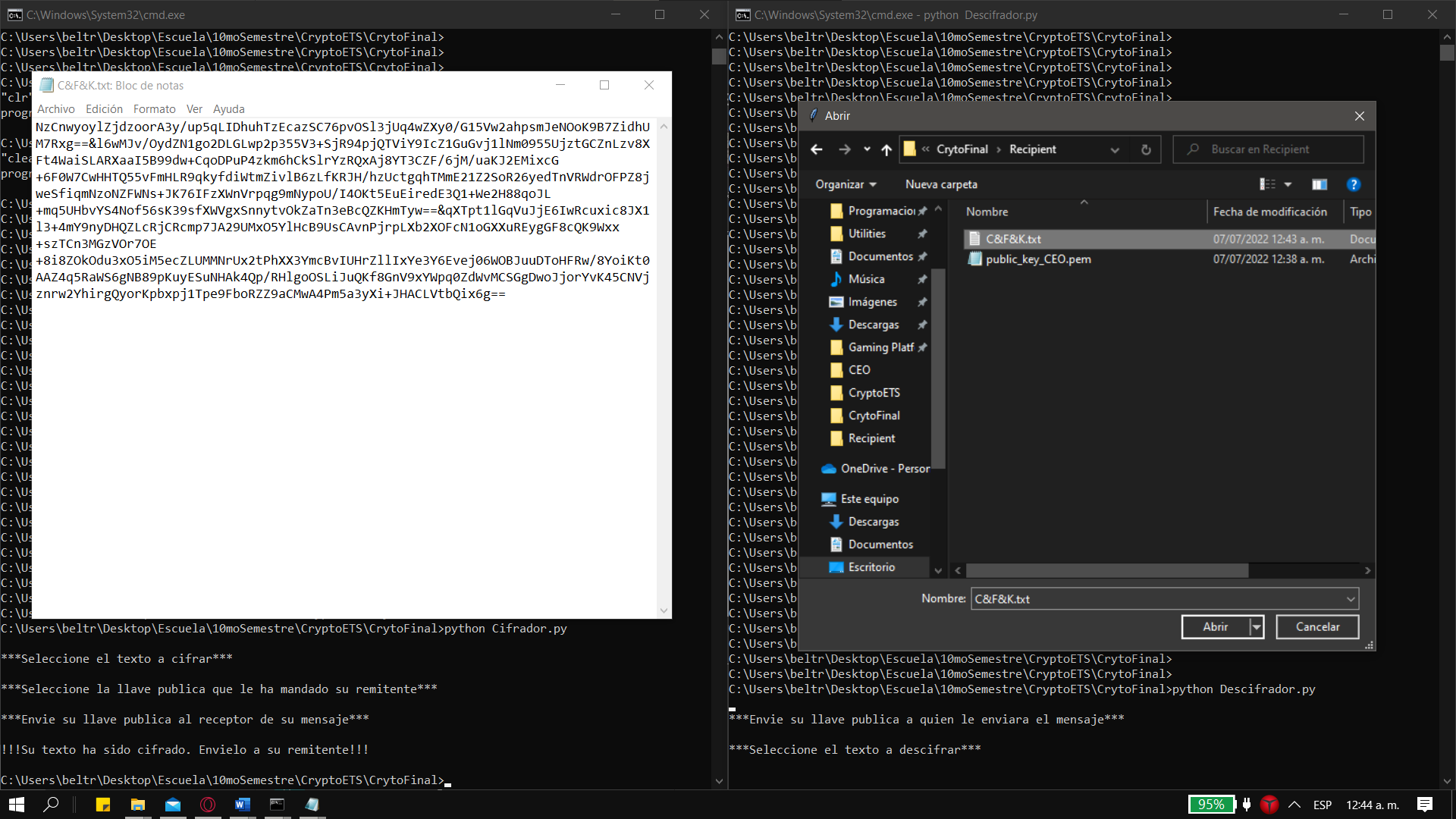
Ahora, del lado del receptor, nos aparece el mensaje “Selección el texto a descifrar”, ahorita no hay ningún texto a descifrar en ningún lado, así que volveremos con el emisor, quien ya tiene en su carpeta la llave publica que le estaba pidiendo Cifrador.py, así que procedemos a seleccionarla y abrirla.



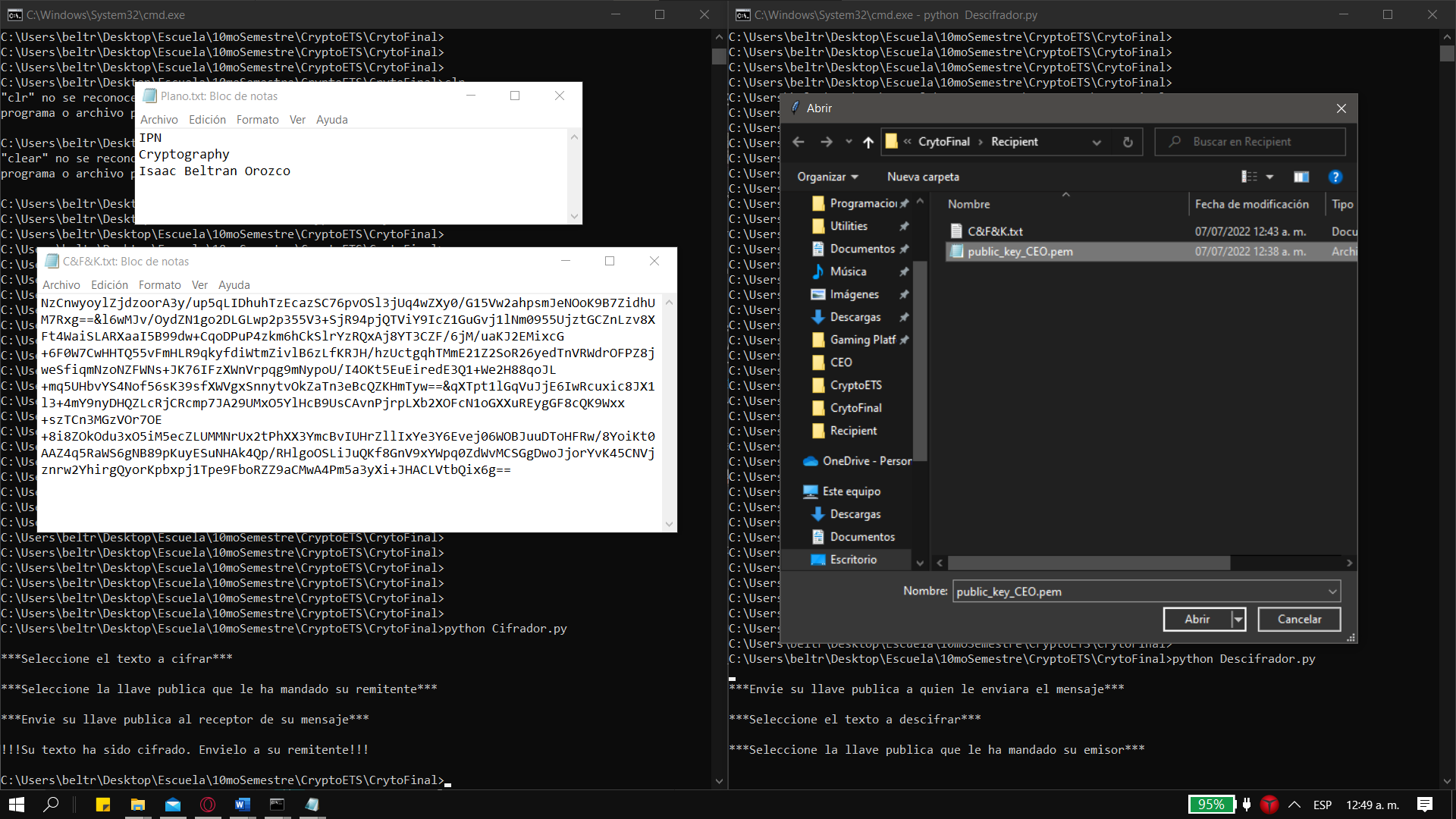
De nuevo, la ventana se cerro al seleccionar la llave y apareció otra, esta vez una para guardar archivos. Y de nuevo parece ser una llave pública. Cambiamos el nombre de la misma para que se sepa que la misma la genero el emisor y “se la pasamos al receptor” guardándola en su carpeta.



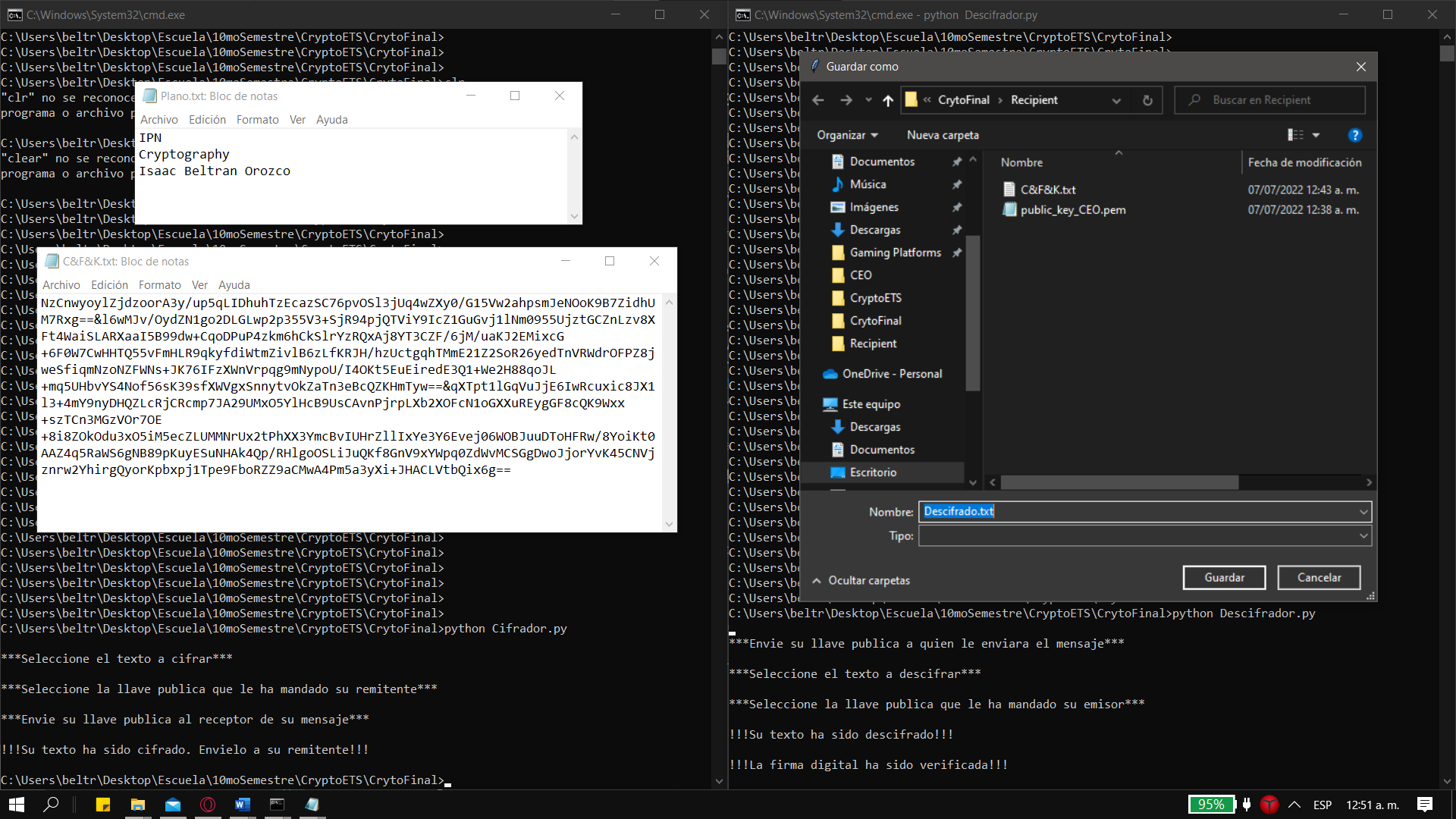
Apareció una nueva ventana para guardar archivos en el lado del emisor, pero esta vez se trata de un txt. El texto en consola dice “Su texto ha sido cifrado. Envíelo a su remitente”. Así pues, procedemos a guardarlo en la carpeta del receptor.



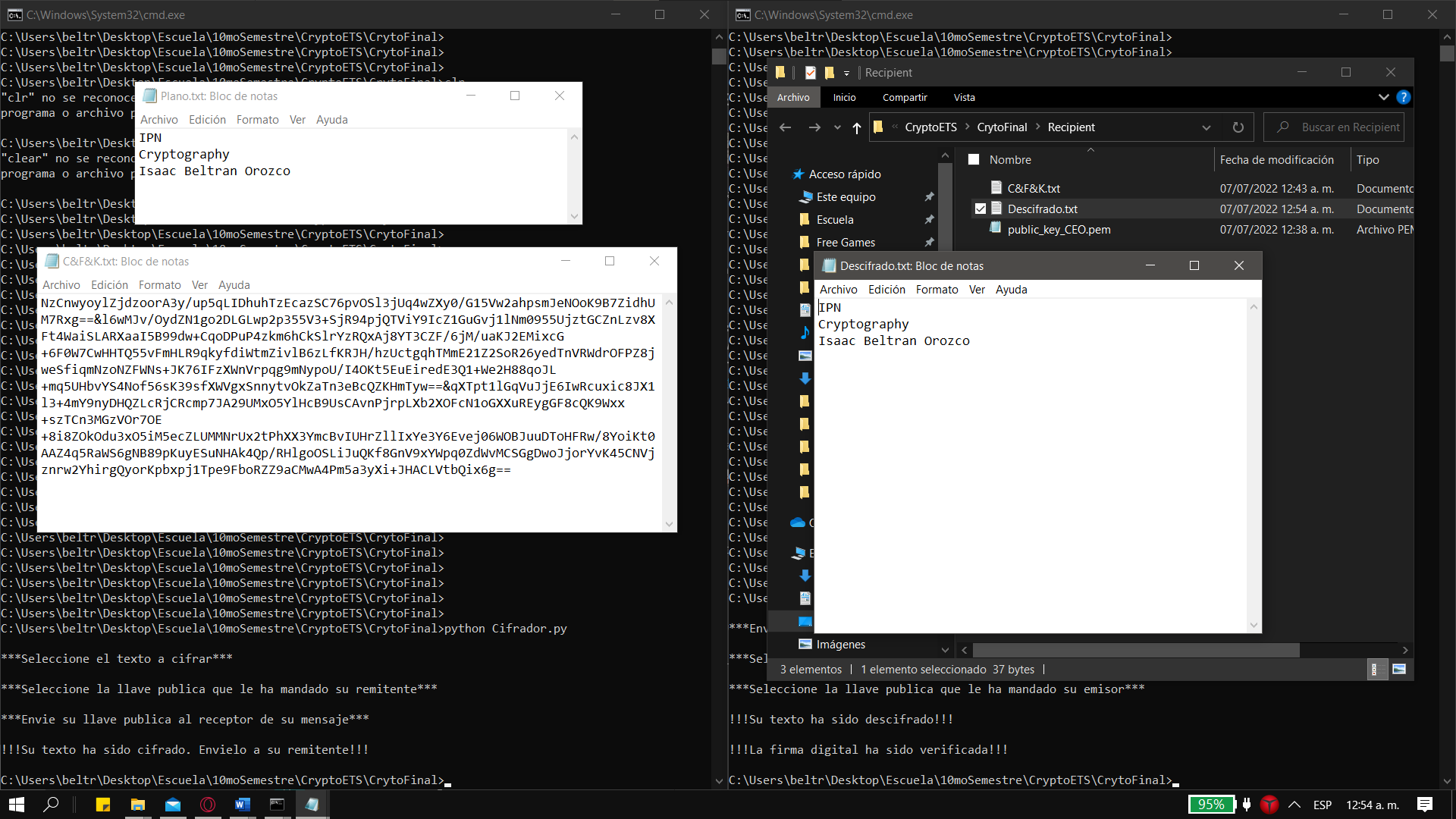
Parece que Cifrador.py ya termino su ejecución. Revisando el txt que envió al receptor vemos que, en efecto, parece no haber mas que caracteres aleatorios. Bien, pues procederemos a seleccionarlo para intentar descifrarlo.



Apareció otra ventana y otro texto que nos indica “Seleccione la llave publica que le ha mandado su emisor”. La seleccionamos de nuestra carpeta.



Y finalmente en la consola aparecen 2 mensajes: “Su texto ha sido descifrado” y “La firma digital ha sido verificada”, ademas de que aparece una ultima ventana de guardar archivo con uno que se llama “Descifrado.txt”. Lo guardamos.



Y al final, el contenido de “Descifrado.txt” es igual al de “Plano.txt”.

**MANUAL DE USUARIO**

**Preliminares**

1. Asegúrese de tener instaladas las bibliotecas tkinter y PyCryptodome antes de correr los programas.
2. Los únicos archivos que se tienen que ejecutar para realizar exitosamente el cifrado y descifrado de un mensaje son Cifrador.py y Descifrador.py respectivamente, ninguno de los demás requiere de previa ejecución. Sin embargo, si hay que remarcar que tanto Cifrador.py como Descrifrador.py dependen de funciones que hay en todos los demás archivos, a saber: file\_admin.py, cifrado\_AES.py, cifrado\_RSA.py y digital\_signature.py; por lo que ambos, por como estan programados, requieren estar en la misma carpeta que todos los archivos mencionados (o con una copia de los mismos), durante todo el tiempo de ejecución preferiblemente. Denotar que, sin embargo, estos no dependen entre sí, por lo que, en teoría, pueden funcionar perfectamente incluso en distintas computadoras, siempre y cuando, primero y de nuevo, tengan en la misma carpeta en donde se encuentren una instancia de cada uno de los mencionados archivos de funciones; y segundo, puedan hacerse llegar entre si las llaves públicas y el archivo cifrado.
3. Le rogamos no pierda de vista las interfaces de tkinter, ya que no aparece en la barra de tareas (al menos en Windows) y puede terminar perdiéndose entre sus ventanas o en el fondo de su escritorio.
4. También le rogamos que, si va a ejecutar ambos archivos, Cifrador y Descifrado, en la misma pantalla, no pierda de vista que terminal de tkinter pertenece a cuál ejecución. Recomendamos ponerlas en lados opuestos del escritorio.
5. Recuerde que en todo momento el programa muestra la instrucción que debe realizar la persona, ya sea emisor o receptor, en la terminal en que se ejecute.

**Ejecución**

1. Como emisor: Ejecute Cifrador.py
2. Como emisor: Seleccione el texto a cifrar.
3. Como receptor: Ejecute Descifrador.py.
4. Como receptor: Envié la llave .pem que acaba de generar a su emisor.
5. Como emisor: Seleccione la llave .pem que su receptor le mando.
6. Como emisor: Envié la llave .pem que acaba de generar a su receptor.
7. Como emisor: Envié el texto cifrado a su receptor.
8. Como receptor: Seleccione el texto a descifrar.
9. Como receptor: Seleccione la llave .pem que su emisor le mando.
10. Como receptor: Guarde el texto descifrado.

**BIBLIOGRAFIA**

1. Silva, V., Flores, R. & Gonzales, M. (2007). *Tópicos Selectos de Criptografía I*. México: CIDETEC-IPN.
2. García, M. (2013). *Implementación del algoritmo de cifrado AES para bajo consumo sobre FPGA*. México: Universidad Carlos III de Madrid.
3. Stinson, D. (2006). *CRYPTOGRAPHY: Theory and practice*. New York: Chapman & Hall/CRC Press.
4. PyCryptodome. (2022). *PyCryptodome*. junio 28, 2022, de PyCryptodome Sitio web: [https://www.pycryptodome.org/en/latest/#](https://www.pycryptodome.org/en/latest/)
5. Universidad Politécnica de Madrid. (2015). *Píldora 30: ¿Cómo se cifra con el algoritmo AES?*. junio 30, 2022, de Universidad Politécnica de Madrid Sitio web: http://www.criptored.upm.es/thoth/index.php#
6. Wang,S. (2019). *The difference in five modes in the AES encryption algorithm.* 30 junio, 2022, de HighGo Sitio web: <https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/>
7. CyberExplore. (2020). *AES CBC, CFB, OFB: Encryption in Python explained easy part 1*. julio 2, 2022, de YouTube…………………………………………………………  
   Sitio web: https://www.youtube.com/watch?v=KRA\_scVTBp0
8. García, M. (2022). *Criptografía y criptoanálisis*. julio 4, 2022, de Blogger Sitio web: <http://mikelgarcialarragan.blogspot.com/p/criptografia-y-criptoanalisis.html>