Nombre del alumno:

Ignacio Ivan Sanchez Pantoja

Número de matrícula:

18108365

Nombre del profesor:

Israel Alejandro Herrera Araiza

Nombre del curso:

Controles Criptográficos De Seguridad

Actividad:

Intercambio de llaves de Diffie-Hellman.

Fecha:

24/05/2024

**Introducción:**

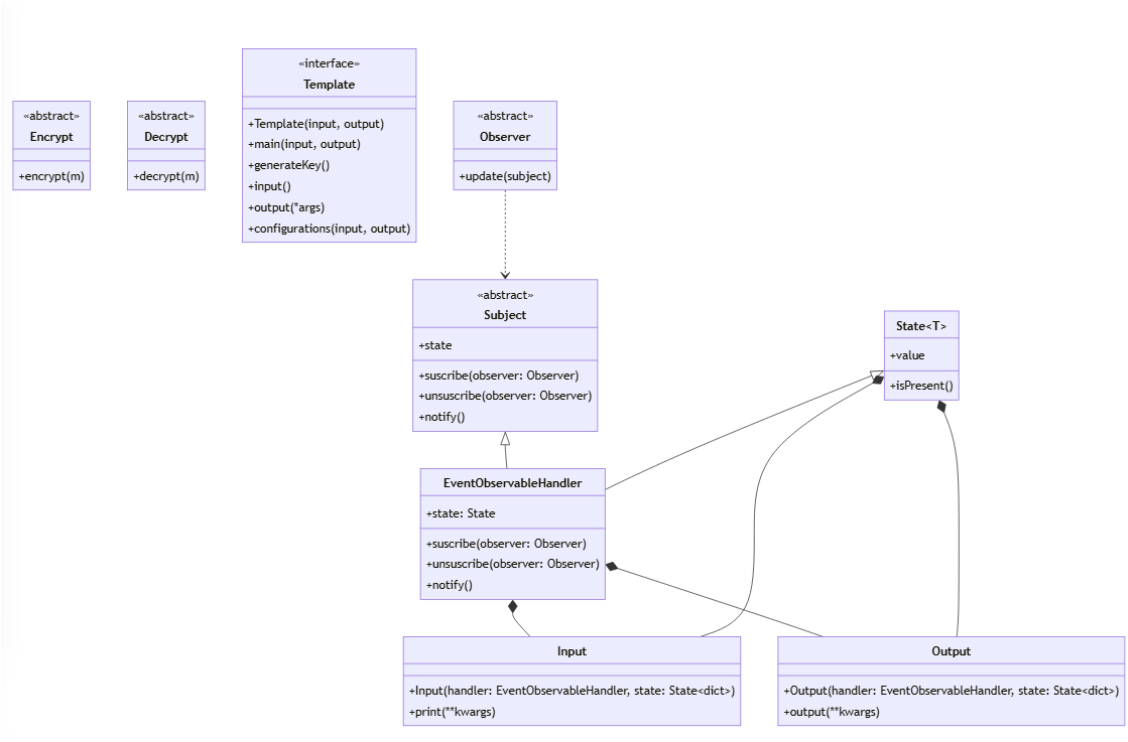
La criptografía es fundamental para la seguridad de la información en la era digital. Dos de los algoritmos más importantes en este campo son RSA y Diffie-Hellman. RSA, desarrollado en 1977, es ampliamente utilizado para el cifrado y la firma digital debido a su robustez y seguridad basada en la factorización de grandes números primos. Por otro lado, el algoritmo Diffie-Hellman, creado en 1976, permite el intercambio seguro de claves a través de canales inseguros, sentando las bases para muchos protocolos de seguridad modernos. La implementación de estos algoritmos en Python y su integración con patrones de diseño como API no solo demuestra su versatilidad, sino también su relevancia continua en la protección de datos sensibles.

**Implementación de algoritmo RSA en Python y integración con el patrón de diseño API**

**Particularidades y consideraciones previas de la aplicación:**

**La arquitectura de la solución es la siguiente:**


la mejor forma para entender la solución presentada es primero acudir a el archivo **kri_cipher.py** este archivo posee la definición de la gran mayoría de patrones de diseño utilizados en el desarrollo del sistema:




Como podemos observar la solución consiste en gran mayoría en el uso de 2 patrones de diseño **Observer** y **Template**, el patrón observer en este caso fue utilizado para poder proporcionar una forma centralizada de entrada y salida de datos es decir una api interna para realizar la compatibilidad entre la vista de una interfaz en base a líneas de comando (CLI) y una interfaz web en este caso por medio del patrón arquitectónico API.

Como es natural la gran mayoría de sistemas utilizan un archivo conocido como el **composition_root,** este archivo tiene el propsito de realizar la inyección de dependencias a los componentes del sistema que lo requieran, generar las instancias de las clases o módulos idóneos para que el sistema siga el core bussines.

En este caso el archivo **composition_root** consiste en 2 funciones 1 utilizada para ejecutar las interfaces en base a CLI y la segunda para poder generar las instancias de los endpoints que conformaran la API.

**Análisis de la interfaz web (composition root web):**

```python
from .RSA.rsa import RSA
from .DiffHellman.diff_h import DiffHellman
from .kri_cipher import Output as OutputHandler
from .kri_cipher import Input as InputHandler
from .kri_cipher import State,EventObservableHandler

from .adapters.drivers.input_adapter_proxy_web import InputAPI,InputAPIObserver
from .adapters.drivens.Output_adapter_proxy_web import OutputAPIAdapter,OutputAPIOb

from .adapters.drivens.Output_adapter_proxy_cli import OutputCLIAdapter
from .adapters.drivers.Input_adapter_proxy_cli import InputCLI

from flask import Flask,request
from flask.views import MethodView
from flask_cors import CORS
from .models.dto.Input import InputRsa,InputDiff
from .models.dto.Output import OutputDto
```

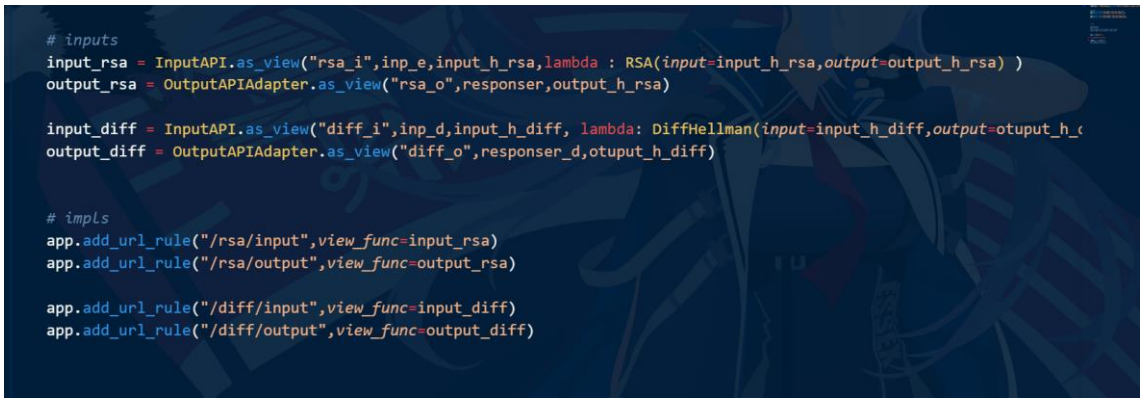
```python
def web_composition_root():
    app = Flask(__name__)
    CORS(app)



    # observers
    output_h_rsa = OutputHandler(handler=EventObservableHandler(State()))
    otuput_h_diff = OutputHandler(handler=EventObservableHandler(State()))
    input_h_rsa = InputHandler(handler=EventObservableHandler(State()))
    input_h_diff = InputHandler(handler=EventObservableHandler(State()))

    dto_rsa = InputRsa()
    dto_diff = InputDiff()
    inp_e = InputAPIObserver(dto_rsa)
    inp_d = InputAPIObserver(dto_diff)
    input_h_rsa.handler.suscribe(inp_e)
    input_h_diff.handler.suscribe(inp_d)
    responser = OutputAPIObserver(dto=OutputDto())
    output_h_rsa.handler.suscribe(responser)
    responser_d = OutputAPIObserver(dto=OutputDto())
    otuput_h_diff.handler.suscribe(responser_d)
```

A grandes rasgos esta sección genera las instancias de las clases usadas como Handlers o Publishers, cada InputHandler o OutputHanlder mediante la relación de composición permite obtener una implementación especifica de algún procesador de eventos o handlers de observables (es decir las clases necesarias para poder enviar o recibir datos dependiendo de la interfaz implementada (sea web o cli); se realizó esto en el caso de que fuera necesario generar una implementación especifica para cada vista o presentación.

Se puede observar que por cada esquema de cifrado se posee un procesador de eventos tanto de entrada como de salida propio, esto es para separar los eventos que en algún momento se puedan enviar a cada clase encargada de procesar la información o en este caso realizar el cifrado de los datos proporcionados por el usuario.

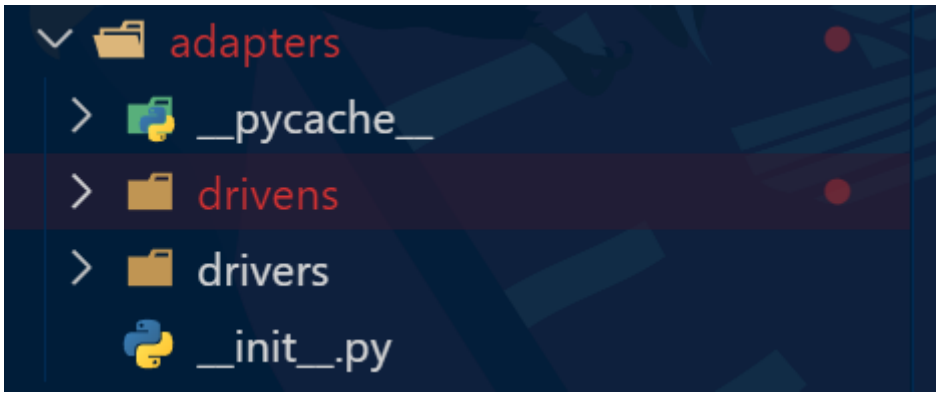
```
# inputs
input_rsa = InputAPI.as_view("rsa_i",inp_e,input_h_rsa,lambda : RSA(input=input_h_rsa,output=output_h_rsa) )
output_rsa = OutputAPIAdapter.as_view("rsa_o",responser,output_h_rsa)

input_diff = InputAPI.as_view("diff_i",inp_d,input_h_diff, lambda: DiffHellman(input=input_h_diff,output=otuput_h_
output_diff = OutputAPIAdapter.as_view("diff_o",responser_d,otuput_h_diff)


# impls
app.add_url_rule("/rsa/input",view_func=input_rsa)
app.add_url_rule("/rsa/output",view_func=output_rsa)

app.add_url_rule("/diff/input",view_func=input_diff)
app.add_url_rule("/diff/output",view_func=output_diff)
```

La clase InputAPI es una implementación de Flask MethodViews, este tipo vistas son usadas para en lugar de procesar plantillas web es decir archivos que serán renderizados y enviados al cliente, estas vistas procesan directamente los verbos HTTP como podrían ser el uso del verbo GET y POST que fueron los que se implementaron en este caso, durante el desarrollo surgió el problema de que las clases que implementan el cifrado necesitan repetidamente generar múltiples instancias de estas mismas para procesar o realizar el cifrado de datos por lo cual se decidió encapsular las dependencias de esta clase en una función anónima o también conocidas como funciones lambda, esta función será invocada cuando la vista reciba datos por parte del usuario por ejemplo al momento de que el usuario reciba una petición get esta función será invocada y se alertaran a su vez a los métodos suscritos a los handlers tanto de entrada como de salida para que las clases  donde se implementan los cifrados puedan recibir los datos y enviar las respuestas JSON.



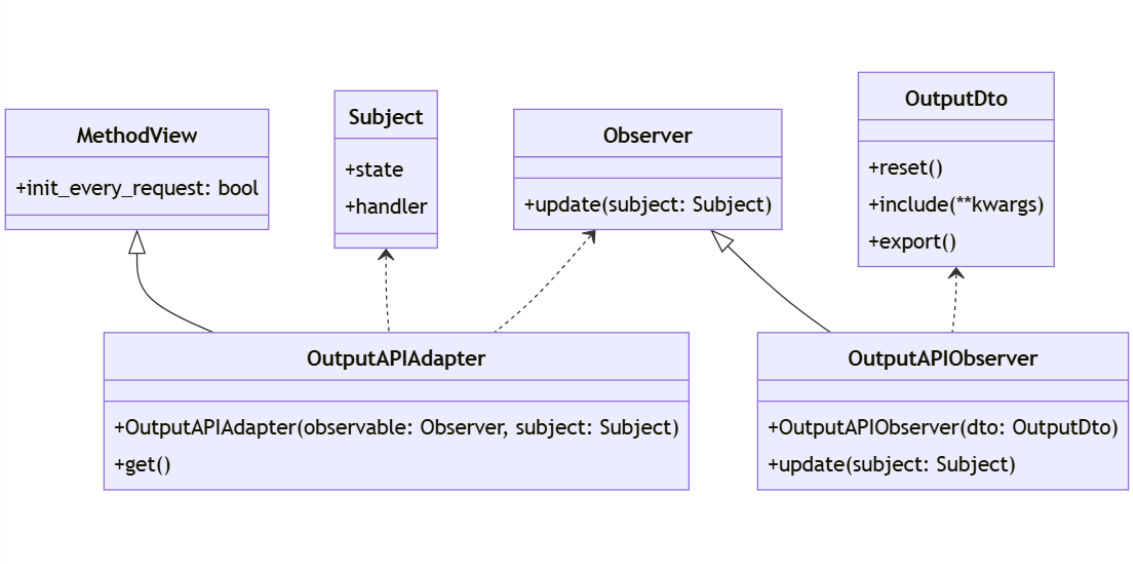
```
# inputs
input_rsa = InputAPI.as_view("rsa_i",inp_e,input_h_rsa,lambda : RSA(input=input_h_rsa,output=output_h_rsa) )
output_rsa = OutputAPIAdapter.as_view("rsa_o",responser,output_h_rsa)

input_diff = InputAPI.as_view("diff_i",inp_d,input_h_diff, lambda: DiffHellman(input=input_h_diff,output=otuput_h_
output_diff = OutputAPIAdapter.as_view("diff_o",responser_d,otuput_h_diff)


# impls
app.add_url_rule("/rsa/input",view_func=input_rsa)
app.add_url_rule("/rsa/output",view_func=output_rsa)

app.add_url_rule("/diff/input",view_func=input_diff)
app.add_url_rule("/diff/output",view_func=output_diff)
```

Con el uso de app.add_url_rule se genera el endpoint que será usado para consultar los datos, la implementación de si se utiliza el verbo POST,GET o cualquier otro depende de la implementación de esta clase, en este sentido, y para poder generar una aplicación escalable, se opto por utilizar el patrón de diseño proxy utilizando los enfoques que define R.C Martin en su obra “Arquitectura limpia”, agrandes rasgos se construyeron adaptadores tanto de entrada como de salida




Generalmente llamados drivers y drivens

Los drivers son aquel conjunto de utilerías necesario para comunicar la aplicación con recursos exteriores (en este caso la implementación de las particularidades específicas de las vistas API o CLI), para explicar este punto podemos ver un diagrama UML de las salidas de la interfaz web:




Podemos observar que OutputAPIAdapter posee únicamente su constructor y su método get el cual no es más que el método usado para procesar las peticiones GET que se dirijan al servidor web.

```python
class OutputAPIAdapter(MethodView):
    init_every_request = False
    def __init__(self,observable:Observer,subject:Subject):
        MethodView.__init__(self)
        self.subject = subject
        self.observable = observable
    def get(self):
        self.observable.dto.reset()
        self.subject.handler.notify()
        return jsonify(self.subject.handler.state.value)
```

Al momento de ser invocado o recibir una petición web este método se encargará de:

1) limpiar los datos almacenados en la anterior petición
2) notificar a los suscriptores que se ha producido un evento es decir que el usuario web ha solicitado información

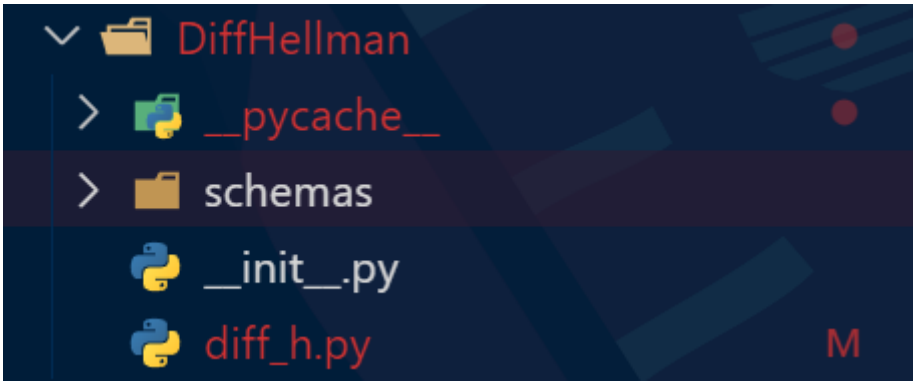
**Implementaciones de RSA y DiffHellman**

La implementación del esquema DiffHellman

Tomando en cuenta las particularidades anteriores podemos ilustrar en un diagrama UML de clases la arquitectura utilizada para la codificación del protocolo DiffHellman creado en 1976.

Las clases RSA y DiffHellman es por mucho una de las clases más sencillas del programa, esto es debido a que cada clase hereda de la interfaz témplate para poder completar lo que sería el patrón de diseño template metohod todas estas particularidades anteriores permiten volver lo más sencillo posible la implementación de dichos algoritmos de cifrado ya que el programa está armado de tal forma en la que simplemente basta con escribir un Script sencillo para poder cifrar los datos independientemente de la interfaz utilizada

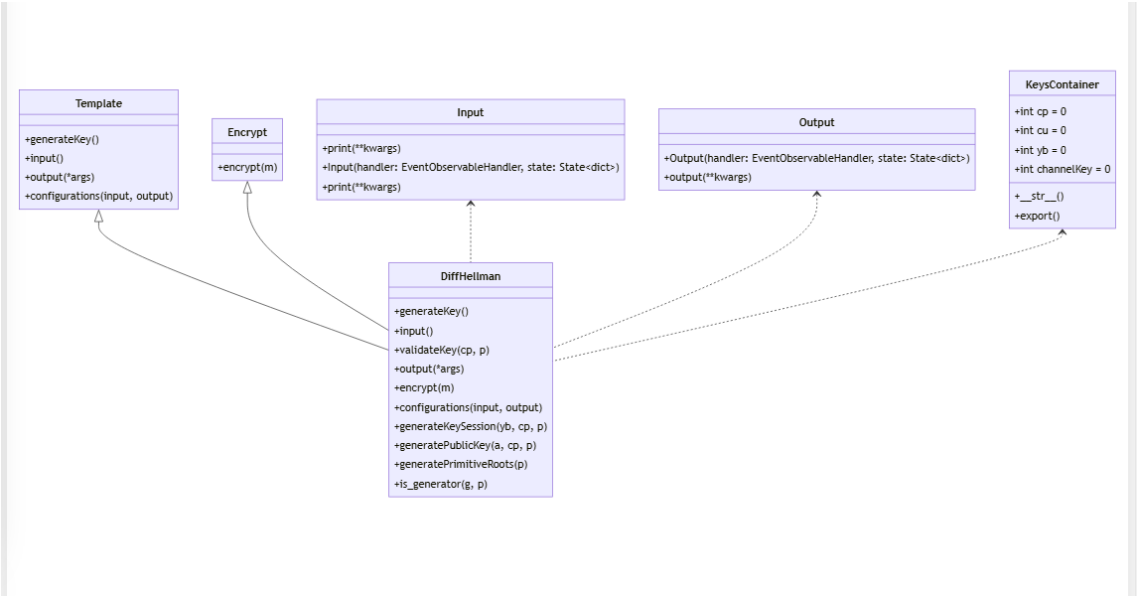
Como podemos observar cualquier algoritmo se conforma por dos estructuras primero un archivo con el nombre del algoritmo y segundo una carpeta con esquemas, también denominado módulo en Python, estos esquemas no son más que simples entidades como Se podrían definir en cuestiones del DDD o diseño conducido por dominio por sus siglas en inglés (domine driven design)




por el ejemplo el algoritmo RSA este algoritmo como menciona su declaración al inicio del programa es únicamente una implementación del libro esto quiere decir que está desaconsejable para su uso en producción esto es debido a que los algoritmos para producción en general, tienen mecanismos para poder llenar o completar los mensajes e igualarlos al tamaño de la clave usada, a estos mecanismos se les conoce como padds.

Algunos ejemplos de estos padds se pueden nombrar al algoritmo PKCS1, como sabemos el proceso del  padding consiste en inyectarle Generalmente bits nulos al mensaje para completarlo y generarle una longitud mayor a la clave utilizada Esto es para que se pueda ampliar el uso del operador modular y de esta forma el atacante no pueda determinar la información que se cifro en base a la longitud de los datos (este ataque es conocido como un ataque de canal lateral y generalmente ocurre debido al problema de que el cifrado RSA es un cifrado determinista).

Sin más dilación la estructura del componente de cifrado RSA es el siguiente:




El patrón template method involucra que se implemente la interfaz en una clase certera para posteriormente implementar una serie de métodos definidos en la interfaz y en el método template el cual es el encargado de ejecutar todos los métodos implementados de la interfaz en un conjunto de pasos ordenados, ahora bien al DiffHellman ser un protocolo y no tener como tal un mecanismo de descifrado, se opto por utilizar el principio de segregación de interfaces definidos en los principios SOLID, de la misma forma que se realizó con la función de cifrado.

**particularidades sobre el funcionamiento del código fuente de la implementación DiffHellman**

absolutamente todos los pasos que seguirá esta y la implementación de RSA se basa en el orden de llamada de las funciones definido en el método main de la interfaz template

```python
class Template(ABC):
    def __init__(self,input,output) -> None:
        self.main(input,output)
    def main(self,input,output):
        # template method
        self.configurations(input,output)
        self.input()
        self.generateKey()
        e = self.encrypt(27)
        self.output(f"[*] mensaje: {27}")
        self.output(f"[*] criptograma: {e}")
        if isinstance(self,Decrypt):
            self.output(f"[*] texto descifrado: {self.decrypt(e)}")
```

- Es decir, primero cargamos las configuraciones especificas de cada cifrado y al mismo tiempo se inyectan las dependencias de observadores de entrada y salida.
- Se invoca el método input para que se recojan todos los valores necesarios para el funcionamiento del cifrado
- se invoca la generación de una clave
- y final mente se cifran los datos, si la clase actual es compatible con la interfaz Decrypt entonces se invocará el método decript para notificar la salida del cifrado
- para fines prácticos se utilizó únicamente el numero 27 como mensaje sin embargo es posible realizar este proceso con una cadena de caracteres o números únicamente empleando un encoder.

Con esto en mente exploremos la implementación de **DiffHellman**:

```python
class DiffHellman(Template,Encrypt):
    def generateKey(self):

        return super().generateKey()
    def input(self):
        self.enrtyPoint.p = int(self._input("p","ingrese un numero primo: "))
        roots = self.generatePrimitiveRoots(self.enrtyPoint.p)
        self.output(roots)

        self.enrtyPoint.a = int(self._input("a","selecciona un numero de la matriz de valores: "))

        self.KEYS_CONTAINER.cp = self.validateKey(int(self._input("cp","coloca la clave privda propia: ")),self.enrtyP
        self.KEYS_CONTAINER.cu = self.generatePublicKey(self.enrtyPoint.a,self.KEYS_CONTAINER.cp,self.enrtyPoint.p)
        self.output(f"clave publica generada: {self.KEYS_CONTAINER.cu} ")
        self.KEYS_CONTAINER.yb  = int(self._input("yb","clave publica del receptor: "))

    def validateKey(self,cp,p):
        if not(cp < p):
            raise ValueError("la clave privada debe ser menor al numero primo seleccionado")
        return cp
    @staticmethod
```

Se define en la función de configuración una forma de rescatar la entrada del usuario, la función input a grandes rasgos será usada para recuperar todos los parámetros necesarios por el programa para poder realizar los cálculos pertinentes.

Adentrándonos en el funcionamiento del protocolo DiffHellman podemos observar el método generatePrimitiveRoots el cual no es mas que

```python
@staticmethod
def generatePrimitiveRoots(p) -> list[int]:
    # regla a < p  donde a es la raiz primitiva
    return [x for x in range(p) if DiffHellman.is_generator(x,p)]
```

La generación de una lista con elementos p-1 que hace uso de una función generadora, esta función generadora no será nada mas ni nada menos que la forma para crear la base que impone la formula del protocolo DiffHellman, en otras palabras:

$$(a^{cp}) mod\ p$$

este numero **a** es posible intercambiarlo por la red al igual que el numero **p** o número primo seleccionado, cp corresponde a la clave privada que posteriormente el usuario seleccionara.

El numero **a** o la base usada debe cumplir 2 propiedades:

- este numero a debe cumplir la regla de que debe ser menor que p forzosamente
- Debe ser un generador del grupo para asegurar que todas las posibles claves se puedan generar en un espacio determinado de claves, tomando en cuenta estos datos la función generador **is_generator**, se asegura que **a** tiene la propiedad de que sus potencias cubren todos los elementos de un grupo discreto sin repetir ningún elemento, en otras palabras esta función asegura que ( a ) es adecuado para ser usado en el intercambio de claves Diffie-Hellman, garantizando que todas las posibles claves pueden ser generadas y, por lo tanto, asegurando la robustez del protocolo.

Veamos un ejemplo del funcionamiento de generatePrimitiveRoots:

Supongamos en este caso que p = 7 y no partiremos desde 1 sino que partiremos desde el numero 3 (esto es por facilidad ya que como sabemos el numero 3 es un numero primo), mencionamos esta sección de código:

```python
@staticmethod
def generatePrimitiveRoots(p) -> list[int]:
    # regla a < p  donde a es la raiz primitiva
    return [x for x in range(p) if DiffHellman.is_generator(x,p)]
```

En otas palabras el funcionamiento actual seria

```
def generatePrimitiveRoots(p) -> list[int]:
    # regla a < p  donde a es la raiz primitiva
    # eq en ejemplo: [x for x in range(3,p) if DiffHellman.is_generator(x,p)]
    return [x for x in range(p) if DiffHellman.is_generator(x,p)]
@staticmethod
```

Entonces se iniciará al item actual (x=3):

1) Se invoca a is_generator(3,7)

En is_generator se realiza el siguiente procedimiento:

Inicialización: Comenzamos con una lista vacía generated = [].

Cálculo de potencias: (is_generator(3,7))

Para ( i = 1 ): ( 3^1 \mod 7 = 3 ). Añadimos 3 a generated.

Para ( i = 2 ): ( 3^2 \mod 7 = 9 \mod 7 = 2 ). Añadimos 2 a generated.

Para ( i = 3 ): ( 3^3 \mod 7 = 27 \mod 7 = 6 ). Añadimos 6 a generated.

Para ( i = 4 ): ( 3^4 \mod 7 = 81 \mod 7 = 4 ). Añadimos 4 a generated.

Para ( i = 5 ): ( 3^5 \mod 7 = 243 \mod 7 = 5 ). Añadimos 5 a generated.

Para ( i = 6 ): ( 3^6 \mod 7 = 729 \mod 7 = 1 ). Añadimos 1 a generated.

Verificación de duplicados: En cada paso, verificamos que el resultado no esté ya en generated. En este caso, todos los valores son únicos.

Verificación final: La lista generated contiene los valores [3, 2, 6, 4, 5, 1], que son todos los elementos del grupo ( {1, 2, 3, 4, 5, 6} ). Como la longitud de generated es ( p-1 ), concluimos que ( g = 3 ) es un generador del grupo multiplicativo de enteros módulo ( p = 7 ).

Otra forma de ver el mismo algoritmo es mediante el siguiente código:

```
>>> for n in range(1,7): {n:set((n**x)%7 for x in range(1,7))}
...
{1: {1}}
{2: {1, 2, 4}}
{3: {1, 2, 3, 4, 5, 6}}
{4: {1, 2, 4}}
{5: {1, 2, 3, 4, 5, 6}}
{6: {1, 6}}
```

Es decir podemos ver que 3 no es el único numero base o a que cumple la propiedad, también es posible tomar el numero 5 para realizar el cálculo.

Esta es la implementación del algoritmo de prueba de generadores, es un algoritmo de generación de números pseudoaleatorios.

Otra forma de realizar este proceso es con el Método de Congruencia Lineal.

```
def input(self):
    self.enrtyPoint.p = int(self._input("p","ingrese un numero primo: "))
    # se enviara por la red
    roots = self.generatePrimitiveRoots(self.enrtyPoint.p)
    # se selecciona un numero a que se comparte en la red

    self.output(roots)

    self.enrtyPoint.a = int(self._input("a","selecciona un numero de la matriz de valores: "))

    self.KEYS_CONTAINER.cp = self.validateKey(int(self._input("cp","coloca la clave privda propia: ")),self.enrtyPoint
    self.KEYS_CONTAINER.cu = self.generatePublicKey(self.enrtyPoint.a,self.KEYS_CONTAINER.cp,self.enrtyPoint.p)
    self.output(f"clave publica generada: {self.KEYS_CONTAINER.cu} ")
    self.KEYS_CONTAINER.yb  = int(self._input("yb","clave publica del receptor: "))
```

Posterior a la generación de las raíces el usuario seleccionara un numero para compartirlo con un segundo como numero base, el self.output método asegura la emisión de estos datos tanto a la interfaz de línea de comandos como a la API.

Posterior a ello recuperamos el numero seleccionado y colocamos la calve privada a utilizar, esta clave privada no es más que el numero al que se la base o el numero a es decir es cp

$$pk = (a^{cp}) mod\ p$$

Se utiliza la función validatekey para asegurarnos que el numero colocado en p sea un numero primo y a sea menor que p.

El realizar este procedimiento arrojara la clave publica que posteriormente se almacenara en una entidad de contenedora de claves (KEYS_CONTAINER)

Posterior a ello se emitirá la clave publica generada y se solicitará una clave pública del receptor para terminar el proceso

Para el establecimiento de la clave común simplemente se utilizará el siguiente código:

```
4       @staticmethod
5       def generateKeySession(yb,cp,p):
6           return (yb**cp)%p
```

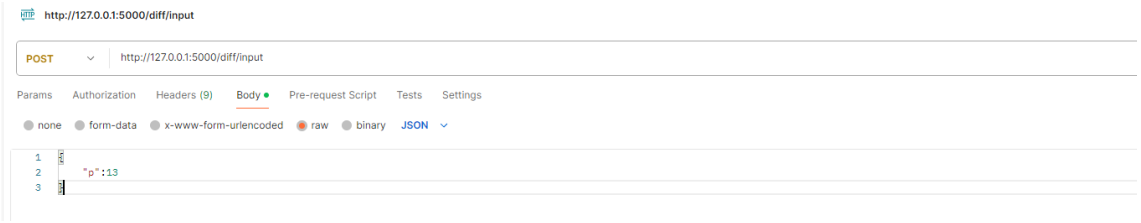
Es decir, la calve del destinatario se utilizara como base y se elevara la clave privada realizando la operación de modular con el numero primo seleccionado, esto arrojara un canal seguro para que ambos usuarios puedan intercambiar datos de forma segura.

POC:

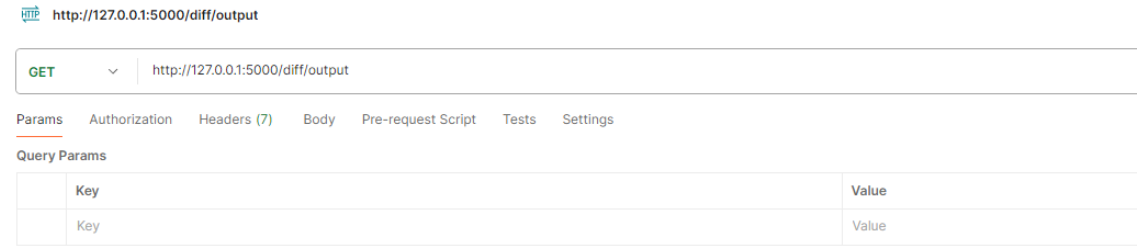
**API WEB:**

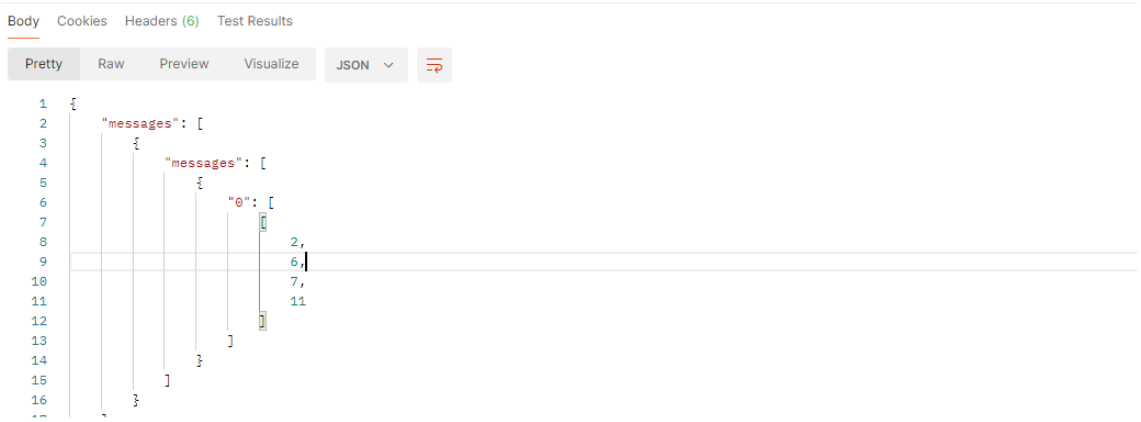
```
PS E:\restore\Cryptographi\KRI_CIPHER> python -m server.composition_root
 * Serving Flask app 'composition_root'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-865-214
```

Utilizando el siguiente documento JSON se primero seleccionará un numero primo:



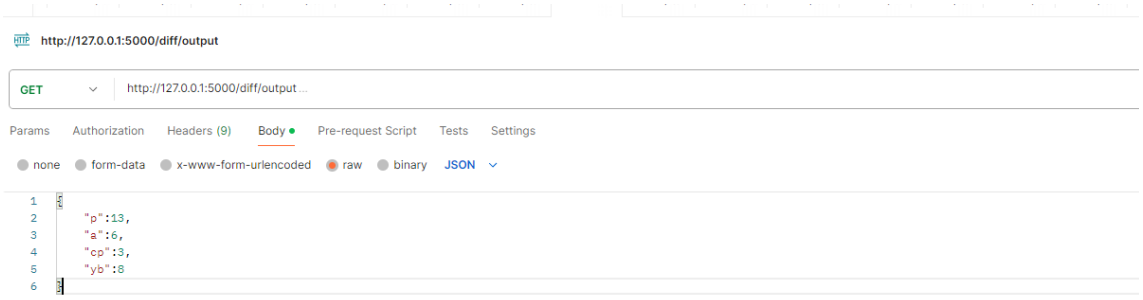

**En el endpoint de salida se generará el siguiente documento json**




Con las raíces generadas.

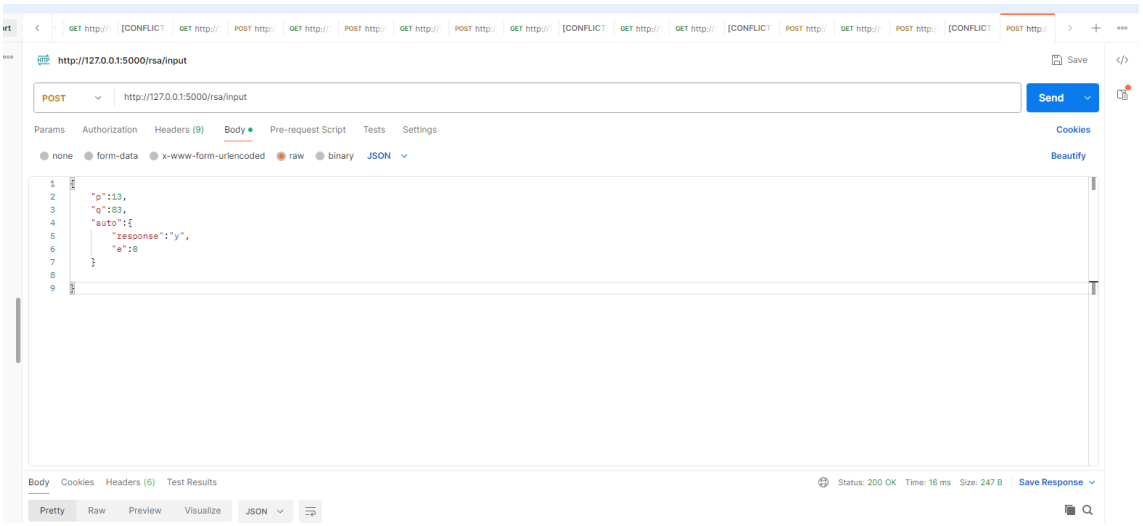
Posterior a ello se introducen la clave privada y pública del homologo a la conexión:




Al final de los procesos e generara una clave publica compartida


```
            },
            {
                "0": [
                    "clave publica generada: 8 "
                ]
            },
            {
                "0": [
                    5
                ]
            },
            {
                "0": [
                    "[*] mensaje: 27"
                ]
            },
            {
                "0": [
                    "[*] criptograma: 5"
                ]
            }
        ]
    }
]
}
```


**RSA:**




**Para el caso del cifrado rsa el endpoint nos requerirá 2 numeros primos y una bandera de si queremos auto generar el numero e, anexando siempre una clave json al numero e, el resultado se puede ver en la siguiente imagen:**


Para probar la interfaz de línea de comandos basta con cambiar la configuración de la llamada del **web_composition_root a cli_composition_root**

```
if __name__ == "__main__":
    #web_composition_root()
    cli_composition_root()
```

**La salida es:**

```
ingrese un numero primo: 13
([2, 6, 7, 11],)
0 ([2, 6, 7, 11],)
selecciona un numero de la matriz de valores: 6
coloca la clave privda propia: 3
('clave publica generada: 8 ',)
0 ('clave publica generada: 8 ',)
clave publica del receptor: 8
('[*] mensaje: 27',)
0 ('[*] mensaje: 27',)
('[*] criptograma: 5',)
0 ('[*] criptograma: 5',)
PS E:\restore\Cryptographi\KRI_CIPHER>
```

En este caso el criptograma corresponde a la clave secreta compartida entre el emisor y el receptor.

## análisis comparativo de la eficiencia y los límites de cada algoritmo

para realizar el análisis de cada algoritmo se decidió realizar una tabla comparativa frente a los algoritmos RSA, AES y ECC:

| Característica | AES (Advanced Encryption Standard) | RSA (Rivest-Shamir-Adleman) | ECC (Elliptic Curve Cryptography) |
|---|---|---|---|
| **Tipo de algoritmo** | simetrico | asimetrico | asimetrico |
| **Longitud de clave común** | 256 | 2048 | 163,256 bits |
| **Seguridad** | Alta seguridad con claves de 256 bits | Alta seguridad pero depende de claves largas (2048 bits mínimo) | Alta seguridad con claves más pequeñas (256 bits de ECC son equivalentes a 3072 bits de RSA) |
| **Rendimiento** | Rápido, especialmente en hardware optimizado (bajo costo computacional) | Lento, especialmente para generación de claves y cifrado (alto costo computacional) | Rápido en comparación con RSA, con claves más cortas y menos recursos computacionales |
| **Uso de recursos** | Son eficientes para el cifrado de grandes volumenes de datos | Consume más recursos debido a claves más largas y operaciones matemáticas complejas | de igual forma que RSA consume mas recursos computacionales para calcular y generar las claves tanto publicas como privadas debido a que hace uso del problema del logratimo discreto asi como las curvas elipticas no singulares, sin embargo su consumo de recursos es significativamente menor que RSA |
| **Tamaño de clave necesario** | Claves cortas (128-256 bits) | Claves largas (mínimo 2048 bits, pero se recomiendan 3072 o más) | Claves cortas (256-521 bits proporcionan alta seguridad) |
| **Tiempo de cifrado/descifrado** | es rapido para grandes volumenes de datos ( sus claves son cortas 256 o 128 bits recomendados ) | Lento para cifrado y descifrado, especialmente en operaciones grandes (cifrado asimétrico más complejo) | Relativamente rápido en comparación con RSA, gracias a la matemática de curvas elípticas |
| **Escalabilidad** | Adecuado para grandes volúmenes de datos | No es adecuado para cifrar grandes volúmenes directamente (se suele combinar con AES) | Similar a RSA, pero más eficiente en uso de claves cortas |
| **Aplicaciones típicas** | Aplicaciones de mensajeria,IPSEC,cifrado de discos | emision de certificados digitales CA | emision de certificados digitales CA, al requerir un coste computacional menor es posible hacer uso de este algoritmo en aplicaciones moviles |
| **Ejemplo de uso práctico** | una red que crea una vpn haciendo uso de GRE y IPSEC, herramientas como veracrypt o LUKS | uso del software PGP para la creacion de certificados CA privados | intercambio de claves en HTTPS, es muy usado en bitcoin |

## Existen varios tipos de ataques que pueden dirigirse a criptosistemas como RSA, ECC y AES. Aquí te dejo una descripción de algunos de los más comunes:

RSA es susceptible a varias vulnerabilidades. Una de las principales es la factorización de números grandes, ya que la seguridad de RSA se basa en la dificultad de factorizar estos números. Si se desarrollan métodos más eficientes para la factorización, la seguridad de RSA podría verse comprometida. Además, los ataques de canal lateral representan otra amenaza significativa. Estos ataques explotan información obtenida del hardware que ejecuta el algoritmo, como el tiempo de ejecución o el consumo de energía, para deducir la clave privada. También existen ataques de relleno de cifrado (padding), que pueden permitir a un atacante descifrar mensajes cifrados si el esquema de relleno no se implementa correctamente, otro método de ataque menos conocido es el uso de el ataque del modulo común, este ataque consiste en que el resultado de la función totiente de Euler (phi) si coincide con otro par publico de claves usado para cifrar un criptograma, entonces no será necesario hacer uso de una clave privada para descifrar los mensajes, como menciona Seth este tipo de ataques demuestran su valía al momento de requerir el recuperado de mensajes antiguos.

Por otro lado, Diffie-Hellman enfrenta vulnerabilidades como los ataques de intermediario (man-in-the-middle). Si las partes no se autentican adecuadamente, un atacante puede

interceptar y modificar las claves intercambiadas, haciéndose pasar por una de las partes. Además, la seguridad de Diffie-Hellman depende de la dificultad de resolver el problema del logaritmo discreto. En grupos pequeños, este problema puede ser más fácil de resolver, comprometiendo la seguridad del protocolo.

Finalmente, AES, aunque es muy seguro, no está exento de vulnerabilidades. Los ataques de canal lateral, similares a los que afectan a RSA, pueden explotar información del hardware que ejecuta AES para deducir la clave de cifrado. Además, aunque AES es resistente a muchos tipos de ataques, un atacante con suficiente poder computacional podría intentar todas las combinaciones posibles de claves mediante ataques de fuerza bruta, especialmente si se utilizan claves más cortas o mal gestionadas.

**Conclusiones:**

La implementación del algoritmo RSA en Python, junto con la integración del patrón de diseño API, demuestra cómo los patrones de diseño Observer y Template pueden centralizar la entrada y salida de datos, facilitando la compatibilidad entre interfaces de línea de comandos (CLI) y web. El uso del archivo composition_root para la inyección de dependencias y la generación de instancias de clases es crucial para mantener la coherencia del sistema.

La implementación de los algoritmos de cifrado RSA y Diffie-Hellman se simplifica mediante el uso del patrón Template Method, permitiendo una estructura modular y adaptable. La encapsulación de dependencias en funciones lambda y el uso de adaptadores de entrada y salida aseguran una aplicación escalable y mantenible.

El análisis comparativo de la eficiencia y los límites de los algoritmos RSA, AES y ECC resalta las vulnerabilidades y fortalezas de cada uno, subrayando la importancia de elegir el algoritmo adecuado según el contexto de uso. La implementación de medidas de seguridad, como el padding en RSA y la validación de claves en Diffie-Hellman, es esencial para proteger contra ataques comunes.

En resumen, la combinación de patrones de diseño robustos y la implementación cuidadosa de algoritmos de cifrado proporciona una base sólida para el desarrollo de aplicaciones seguras y eficientes.

# Referencias


Azad, A.-S. K. (2015). *PRACTICAL CRYPTOGRAPHY*. Taylor Francis Group.

Friedman, W. F. (1987). *THE INDEX OF COINCIDENCE AND ITS APPLICATIONS IN CRYPTANALYSIS*. Laguna Hills, California 92654: AEGEAN PARK PRESS.

Seth James Nielson, C. K. (2019). *Practical Cryptography in Python*. Texas: Apress.

*Anderson, R. (2020). *Security Engineering: A Guide to Building Dependable Distributed Systems* (3rd ed.). Wiley.*

*Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (2018). *Handbook of Applied Cryptography*. CRC Press*