**Explicación técnica: Explotación de vulnerabilidad *export-grade* en protocolo para intercambio de llaves Diffie-Hellman**



**Autor:**

* José Manuel Rodríguez Vázquez – 746193

[**Hipótesis 2**](#_q5ju80c4zp69)

[**Métodos y algoritmos empleados durante el proyecto 2**](#_mv0csqxm0mju)

[1- Fuerza bruta 2](#_nufqe66hjz0m)

[2- Baby steps -> Giant steps 3](#_5w9bw3ttcm8q)

[3- LogJam 4](#_fo1e41es6txr)

[4- Pollard's Rho - Enfoque aleatorio 6](#_m9g7ueciui53)

[**Resultados 10**](#_d6fe3rahr2yc)

[**Conclusiones 14**](#_44jg7p84b3oq)

[**Referencias 15**](#_d4nsa8e6hgun)

# **Hipótesis**

El problema que existe en el intercambio de claves Diffie-Hellman, utilizado en la criptografía representa un problema bastante grave, varias técnicas a probar como Fuerza bruta, LogJam, Baby steps -> Giant steps y Pollard’s Rho, se pueden utilizar para vulnerar este protocolo de intercambio de llaves, las hipótesis relacionadas a las técnicas con las siguientes:

* **Fuerza bruta:** Con la evolución del poder computacional a lo largo del tiempo, un ataque de fuerza bruta se vuelve cada vez más viable para romper estos protocolos.
* **LogJam:** Al bajar de nivel la negociación de clave, podemos utilizar de forma más sencilla el problema de logaritmo discreto para obtener el exponente.
* **Baby steps -> Giant steps:** Este algoritmo tiene una complejidad temporal , lo que puede hacer más eficiente el proceso de un ataque de fuerza bruta.
* **Pollard 's Rho:** Puede resultar más eficiente debido a la aleatoriedad que maneja, aunque, puede durar mucho en ciertas ocasiones.

# **Métodos y algoritmos empleados durante el proyecto**

### **1- Fuerza bruta**

Método bastante sencillo de aplicar, el cual tuvo como base la iteración entre un rango , intercalando los valores de este por el exponente secreto x que genera A, de tal manera que [ ]

Una implementación básica de este algoritmo logró expresarse de la siguiente manera:

p = "0xde26ab651b92a129"

g = 2

A = str(input("Intercepted A: "))

p\_i=int(p, 16)

A\_i=int(A, 16)

B = str(input("Intercepted B: "))

B\_i=int(B, 16)

for i in range(1, p-1):

secret1=pow(g, i, p\_i)

if secret1==A\_i:

print("[+] Clave conjunta encontrada")

secret=i

break

if secret is None:

print("[!] Clave conjunta no encontrada")

shared\_secret=pow(B\_i, secret, p\_i)

### **2- Baby steps → Giant steps**

Algoritmo basado en la implementación de una lista, arreglo, etc, la cual aumenta su tamaño y valores dentro a través de iteraciones [] (Baby steps), aumentando su eficiencia al reducir considerablemente el rango de búsqueda.

Estos valores almacenados, a su vez son comparados por un valor que crece de manera exponencial (Giant Steps) a través de la fórmula []. Cuando los valores coinciden, se habrá encontrado el exponente tal que [ ].

Al trabajar con un número entero de 64 bits, la lista empleada crece de manera exponencial, por lo que se optó por almacenar estos valores directamente en un archivo dentro del almacenamiento físico, en lugar de dejarlos en la memoria RAM.

Se empleó de la siguiente manera:

def baby\_steps(p, g, A, root\_p):

value = 1

baby\_steps\_dict = {}

with open("baby\_steps.txt", "w") as f:

for x in range(root\_p):

f.write(f"{value} {x}\n")

value = (value \* g) % p

def gigant\_steps(p, g, A, root\_p):

gigant\_steps = pow(pow(g, root\_p, p), -1, p)

value = A

with open("baby\_steps.txt", "r") as f:

for line in f:

baby\_value, baby\_x = map(int, line.strip().split())

if value == baby\_value:

return int(baby\_x)

value = (value \* gigant\_steps) % p

return -1

A\_i = int("0x5cda3a2dc931af7e", 16)

p\_i = int("0xde26ab651b92a129", 16)

g = 2

B\_i = int("0x2e9ebd2b01b84a78", 16)

print("[-] Calculando raiz de p...")

root\_p = int(math.ceil(math.sqrt(p\_i - 1)))

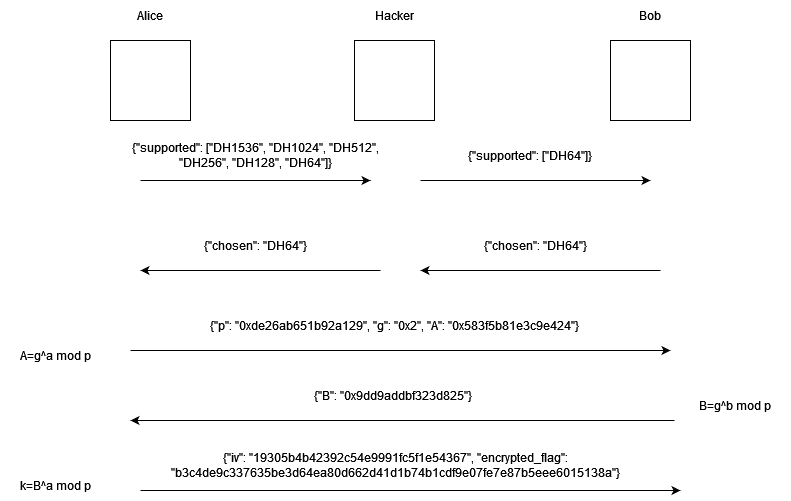
print("[+] Raiz de p encontrada: ", root\_p)

secret = gigant\_steps(p\_i, g, A\_i, root\_p)

print(f"Secret found: {secret}")

### **3- Logaritmo discreto**

Este ataque utiliza logaritmo, discreto para poder encontrar el exponente y con ello poder conseguir la llave que está siendo pasada mediante Diffie-Hellman, gráficamente podemos ver que el ataque consiste en:



Lo que como MITM logramos interceptar es simplemente elegir la versión de Diffie-Hellman a utilizar, elegimos la más débil y con ello podemos lograr solamente ver lo que se manda, así que debemos obtener el exponente, para ello usamos sencillamente las matemáticas:

Por lo tanto realizamos el despeje con logaritmo:

Para ello utilizamos el siguiente código haciendo uso de la librería “sympy”, que lleva a cabo tareas de álgebra computacional, el código queda de la siguiente forma:

*import* math

*import* hashlib

*from* Crypto.Cipher *import* AES

*from* Crypto.Util.Padding *import* unpad

*from* sympy.ntheory *import* discrete\_log

encrypted\_flag="aed26e89092693ee4266b676e1eac3a5805df0e34ab3b17430b1e6ada72754df"

phex="0xde26ab651b92a129"

ghex="0x2"

Ahex="0xb727cdc5e6c9a262"

Bhex="0x8cf4521fba2fd1ea"

iv=bytes.fromhex("c62454a245526136d41557b035310700")

p = int(phex, 16)

g = int(ghex, 16)

A = int(Ahex, 16)

B = int(Bhex, 16)

encrypted\_flag = bytes.fromhex(encrypted\_flag)

def decrypt\_flag(*shared\_secret*: int, *iv*: str, *ciphertext*: str):

sha1 = hashlib.sha1()

sha1.update(str(*shared\_secret*).encode('ascii'))

key = sha1.digest()[:16]

aes = AES.new(key, AES.MODE\_CBC, *iv*)

plaintext = aes.decrypt(*ciphertext*)

*return* plaintext

def DLP(*g*, *A*, *p*):

*return* discrete\_log(*p*, *A*, *g*)

a = DLP(g, A, p)

shared\_secret=pow(B,a,p)

text = decrypt\_flag(shared\_secret,iv,encrypted\_flag)

print(text)

### 

### **4- Algoritmo de enfoque aleatorio**

Este algoritmo es el más complejo que se trató en la investigación. Se trató de una mezcla entre el algoritmo Rho de Pollard y el Algoritmo de Richard P. Brent para la detección de ciclos. En este planteamiento nos centramos en ***dos elementos*** que partirán desde la misma posición hasta que en algún punto aleatorio, colisionen para reducir los rangos de búsqueda Este algoritmo se puede dividir en cinco etapas:

1. ***Pocisionamiento de origen:***

En esta fase, se asignan dos números aleatorios en representación de A y B (a\_rand, b\_rand respectivamente) que varían en un rango de . Estos números cumplen la función para conformar dos conjuntos de tres partes , las cuales son:

Si **‘y’** cumple ciertos requisitos dentro del ciclo de colisión, alterará su valor, al igual que el de **a** y **b** respectivamente.

A ambos conjuntos con el mismo valor inicial, les llamaremos **tortuga** y **liebre**

1. ***Ciclo de colisión:***

Durante esta iteración, se modificarán los valores de dependiendo del resultado de . Se elige el número 3 en este algoritmo debido a que es un número chico que simplifica los cálculos, pero podría ser cualquier primo que eligiéramos.

Si :

Si :

*Cualquier otro residuo resultará en:*

Por lógica de aleatoriedad, en algún punto la **‘y’** de ambos elementos **tortuga** y **liebre**.

serán iguales.

1. ***Reasignación de valores iniciales post-bucle:***

Se toman en segundo y tercer valor de ambos elementos de la siguiente manera:

**Tortuga:** aT (segundo valor), bT (tercer valor)

**Liebre:** aL (segundo valor), bT (tercer valor)

Con estos valores, recalcularemos a y b nuevamente:

1. ***Reducción de valores***

En este punto del algoritmo, se inicializa un valor **‘d’**, generado a través del Máximo Común Divisor entre y .

Este MCD es el valor real de **d**, con el cual normalizamos los valores **a, b** a un rango más corto

Los valores normalizados de a,b se calculan con el valor entero de la división entre **d**, de la siguiente manera:

De igual manera, normalizamos el módulo actual (p), para reducir todos los valores posibles. Este se calculará como

\_

1. **Calcular candidatos de k que genere A**

Generamos una a partir del inverso modular de b en el nuevo módulo, aplicando al inverso de b el módulo nuevamente

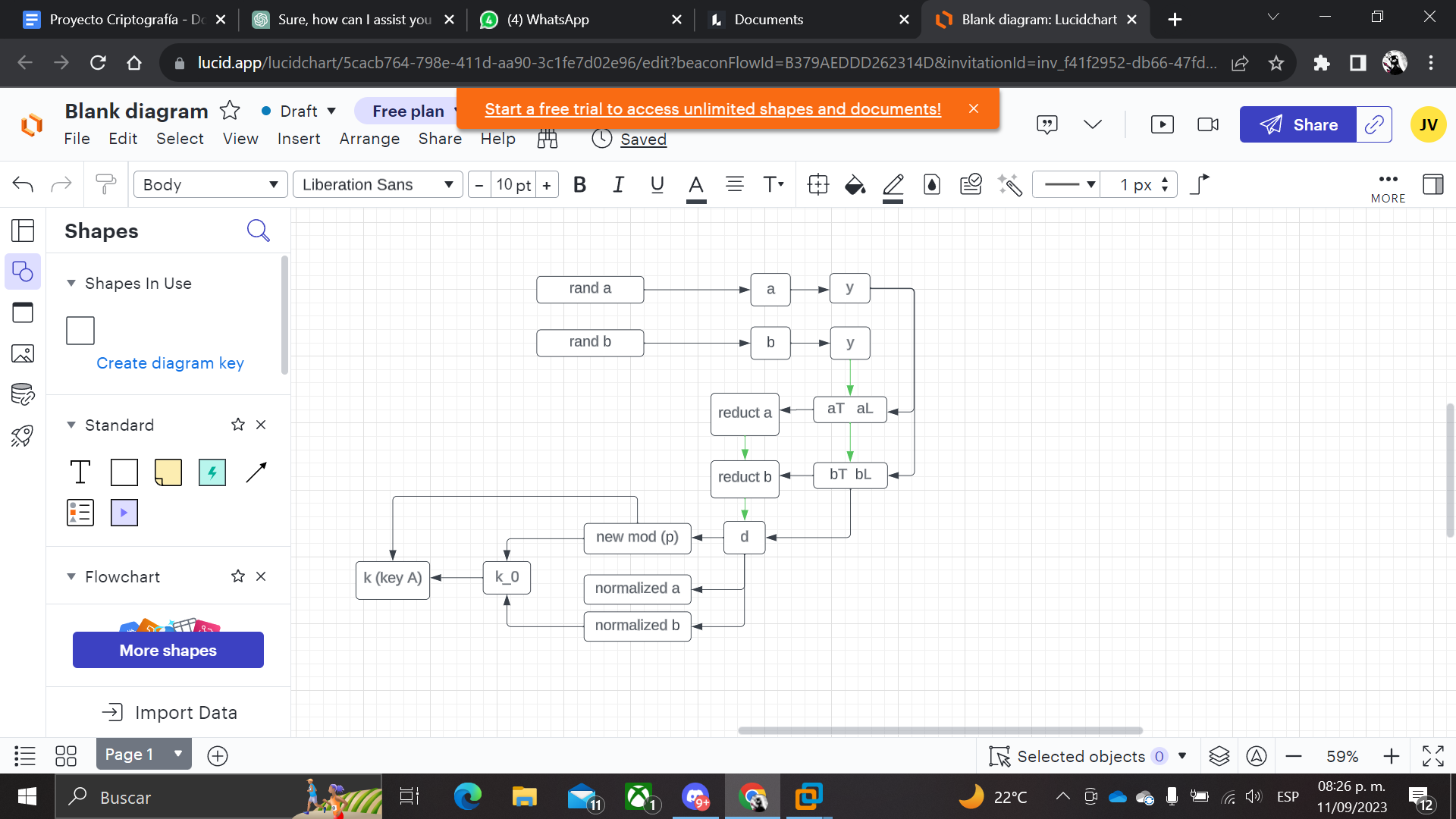
*nn*

Y con k sub-zero calculada, iteramos aumentando el valor de n\_mod una vez por cada ciclo, es decir, calculamos k aumentando el valor de en *n\_* pasos

\_

Por lo que, dependiendo del número par en el que partamos, generamos más o menos rápido el valor que genera A.

De manera resumida, los valores se generan de la siguiente manera:



El código del algoritmo quedaría planteado de la siguiente manera:

A = "0x5cda3a2dc931af7e"

p = "0xde26ab651b92a129"

g = 2

B = "0x2e9ebd2b01b84a78"

p = int(p, 16)

A = int(A, 16)

B = int(B, 16)

print(f'\t--> Parametros: (p, g, A) = ({p}, {g}, {A})')

def f(runner):

y, a, b = runner

if y%3 == 0:

y = g\*y %p

a += 1

elif y%3 == 1:

y = A\*y %p

b += 1

else:

y = y\*y %p

a \*= 2

b \*= 2

return y, a%(p-1), b%(p-1)

a\_rand = random.randint(1, p-2)

b\_rand = random.randint(1, p-2)

liebre = tortuga = (pow(g, a\_rand, p)\*pow(A, b\_rand, p) % p, a\_rand, b\_rand)

while True:

tortuga = f(tortuga)

liebre = f(f(liebre))

if tortuga[0] == liebre[0]:

break

aT, bT = tortuga[1:]

aL, bL = liebre[1:]

a = aL - aT

b = bT - bL

d = math.gcd(bL-bT, p-1) #mcd entre ambos valores, para posible d

a = a//d #Normalizar valor a

b = b//d #Normalizar valor b

new\_mod = (p-1)//d #Normalizar nuevo modulo

k0 = a \* pow(b, -1, new\_mod) % new\_mod

print(f'[-] Revisando posible candidato para la clave d = {d}')

for \_ in range(d): # \_ es d-1

k = k0 + \_ \* new\_mod # calcula k desde el inicio k0 + d-1 + el nuevo primo

print(f'\t--> For k = {k}, g^k == {pow(g,k,p)} modulo {p}.') # traza

if pow(g,k,p) == A: #Vemos que la clave funcione correctamente para generar A

print(f'[+] AVISO: Clave compartida encontrada: {k}')

break

### 

### **Resultados**

Para automatizar los resultados que funcionaron, utilizamos un script en bash para automatizar las conexiones y pasar las variables al código de python, se logró de la siguiente manera:

*#!/bin/bash*

function ctrl\_c() {

echo -e "${red}\n\n[!] Saliendo...\n${normal}"

rm $py\_file 2>/dev/null

rm out.txt 2>/dev/null

exit 1

}

*# Ctrl+c*

trap ctrl\_c INT

red='\033[1;31m'

gray='\033[1;30m'

normal='\033[0m'

*########## Banners ##########*

py\_file=LogJam.py

target="socket.cryptohack.org 13379"

echo -ne "[+] Estableciendo conexión a ${target}"

echo

tempfile=$(mktemp)

{

echo -ne '{"supported": ["DH64"]}\n';

sleep 2;

echo -ne '{"chosen": "DH64"}\n';

sleep 2;

} | nc $target >$tempfile &

nc\_pid=$!

received\_data=false

wait $nc\_pid

> out.txt

*while* read -r line; do

echo "$line" | tee -a out.txt

*if* [[ "$line" == \*'"encrypted\_flag"'\* ]]; then

received\_data=true

*break*

fi

done <$tempfile

rm $tempfile

*if* $received\_data; then

echo -ne "\n[+] Comunicación terminada" | tee -a out.txt

*else*

echo "[!] No se recibieron datos"

fi

*#Valores*

A=$(cat out.txt | grep -o '"A": "0x[^"]\*' | cut -d'"' -f4)

iv=$(cat out.txt | grep -o '"iv": "[^"]\*' | awk -F ': ' '{print $2}' | tr -d '"')

encrypted\_flag=$(cat out.txt | grep -o '"encrypted\_flag": "[^"]\*' | awk -F ': ' '{print $2}' | tr -d '"')

B=$(cat out.txt | grep -o '"B": "0x[^"]\*' | cut -d'"' -f4)

p=$(cat out.txt | grep -o '"p": "0x[^"]\*' | cut -d'"' -f4)

py\_load=$(cat << EOL

)

*######################## Código de python con variables de bash ########################*

EOL

)

echo "$py\_load" > "$py\_file"

python3 "$py\_file"

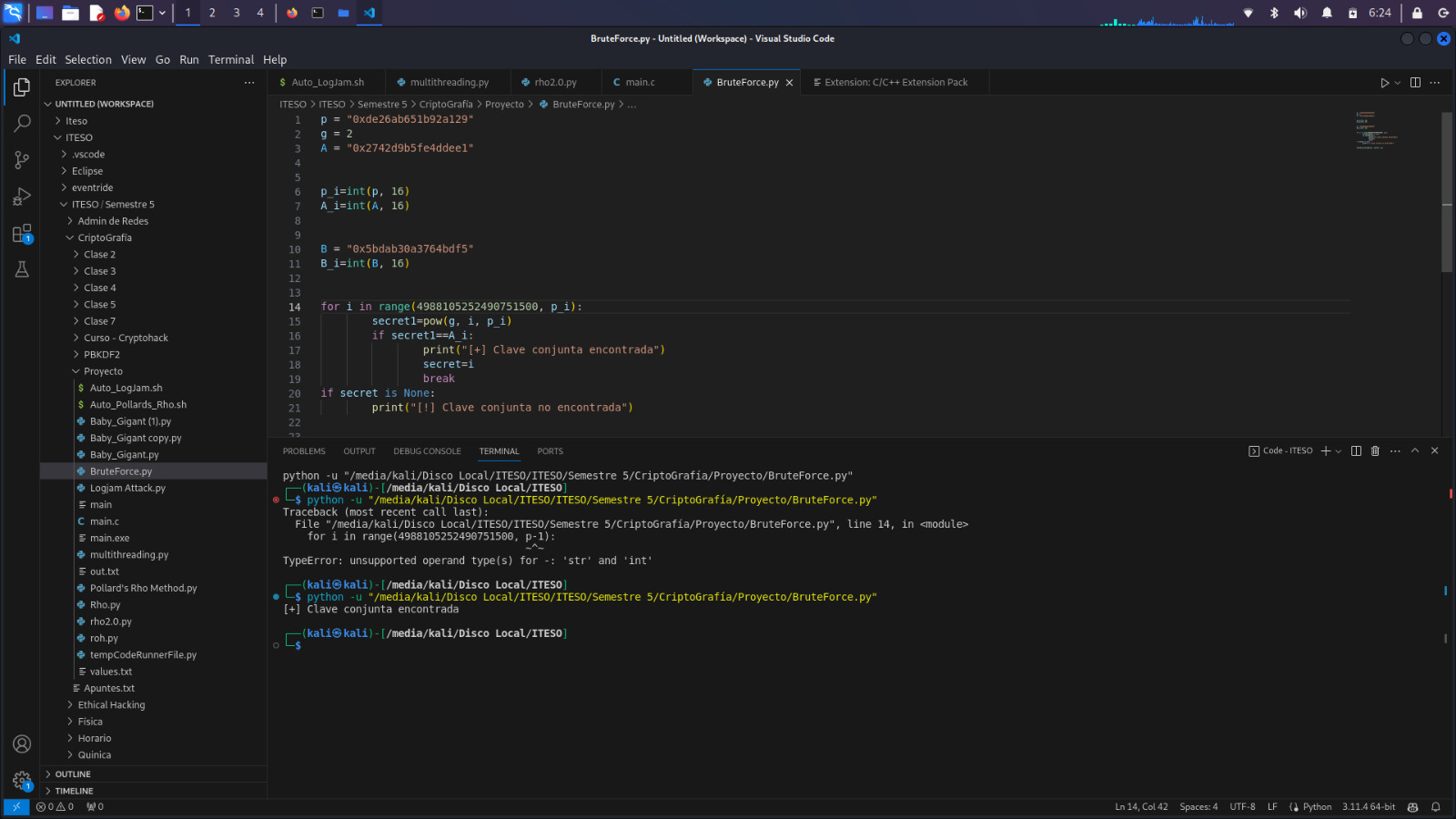
rm "$py\_file"

rm out.txt

* **Fuerza bruta:**

Este método deja de ser óptimo, ya que tendríamos que disponer de por lo menos 1 millón de años, por lo que nos vimos obligados a buscar otros métodos para “atacar” este problema.

Sin embargo, comprobamos que funciona al poner un rango cercano que ya conocíamos, y encontrar la clave generadora de A.



* **Logaritmo:**

Es un método de “ataque” a AES, por lo que

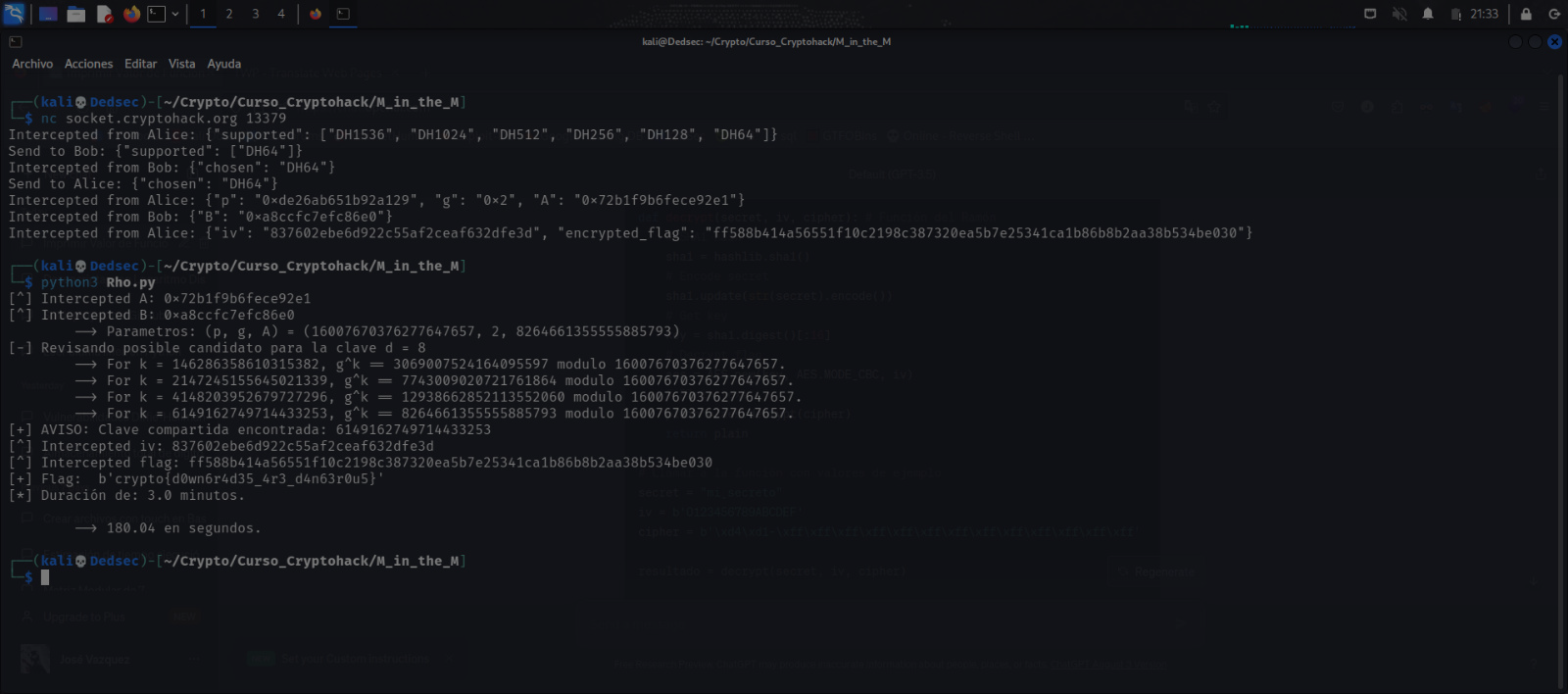
Se llevó a cabo la automatización del script para que todas las llaves y cosas a interceptar se hicieran automáticamente, para ello se hizo uso de una combinación de bash y python.

****

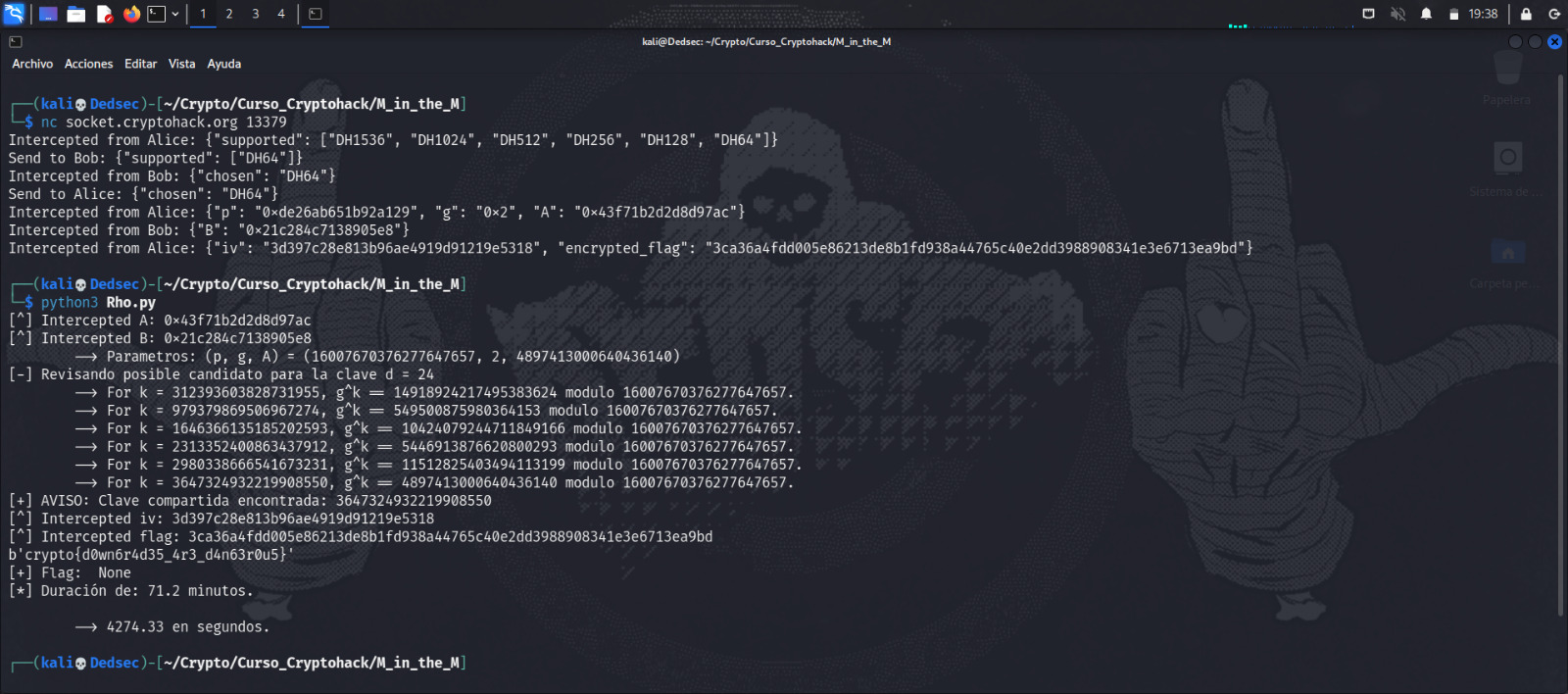
* **Ataque de semilla aleatoria:**

Se demostró que la aleatoriedad del programa puede ser tanto beneficiosa como problemática, al rondar tiempos de ejecución que podían ir desde los 3 minutos hasta las 2 horas. Pero siempre oscilando en un promedio de entre 30 a 40 minutos, tanto para 64 como para 128 bits:

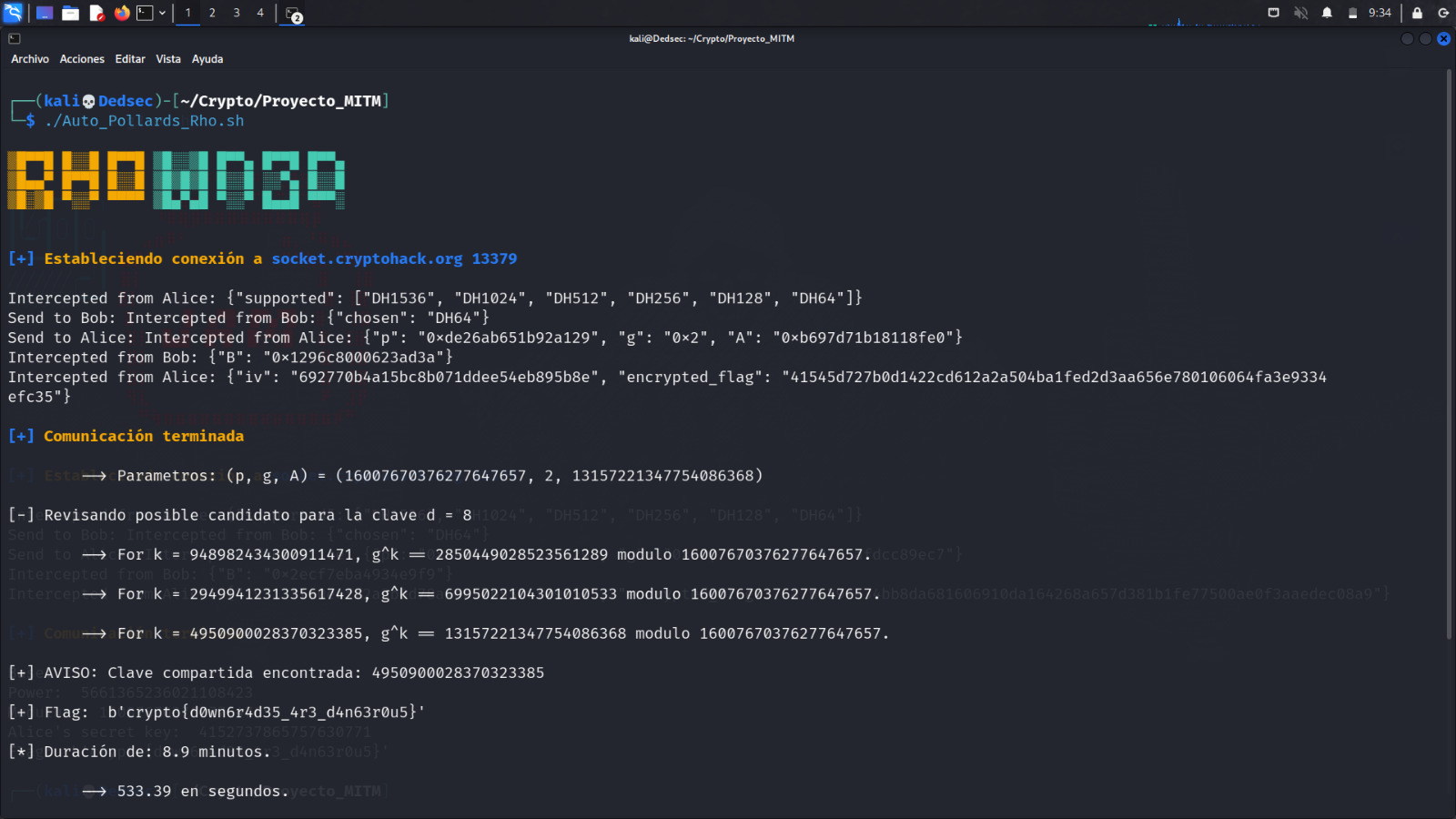
* 3 minutos con 64 bits:



* 71 minutos con 64 bits:



* 8 minutos con 128 bits:

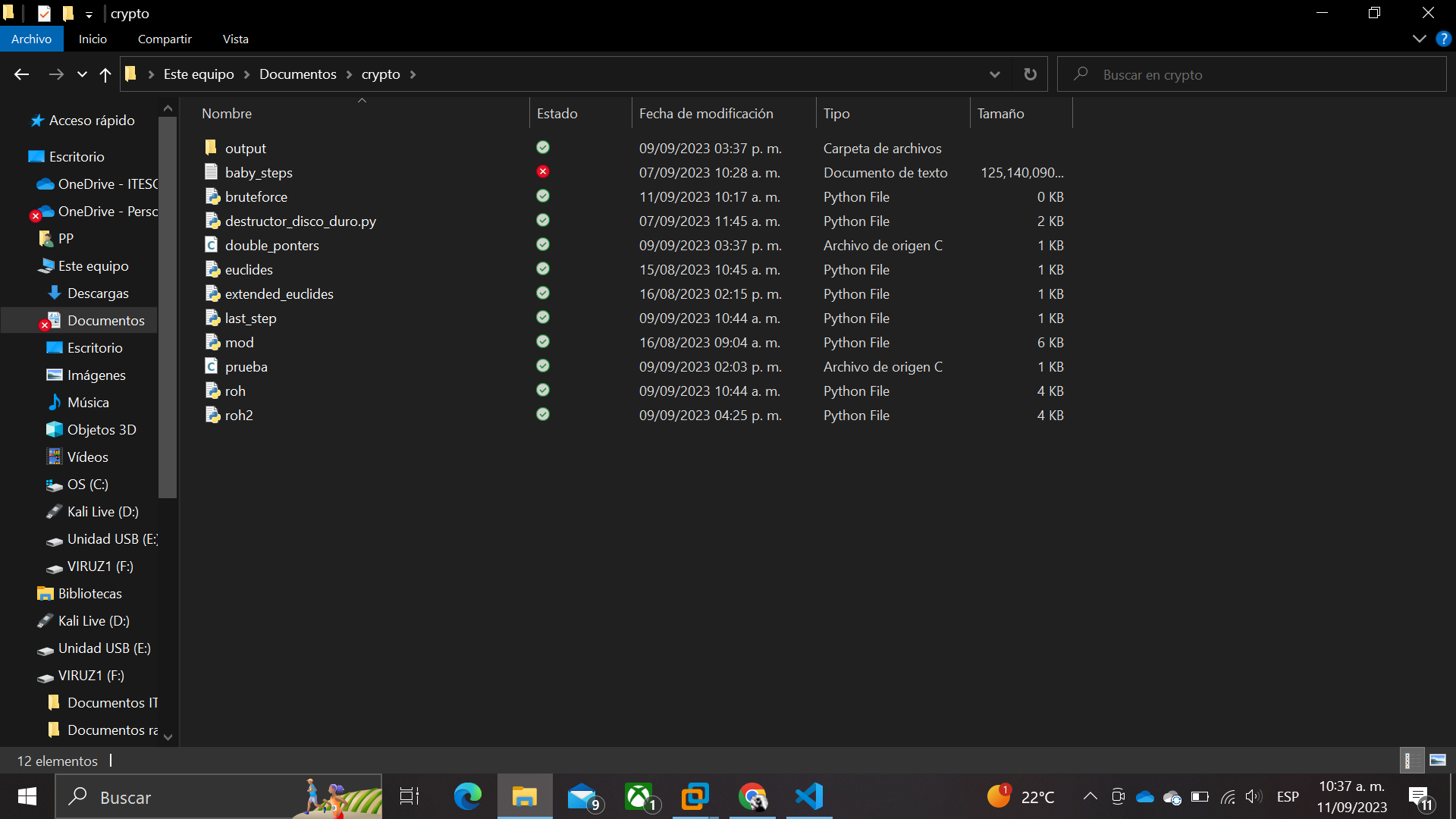


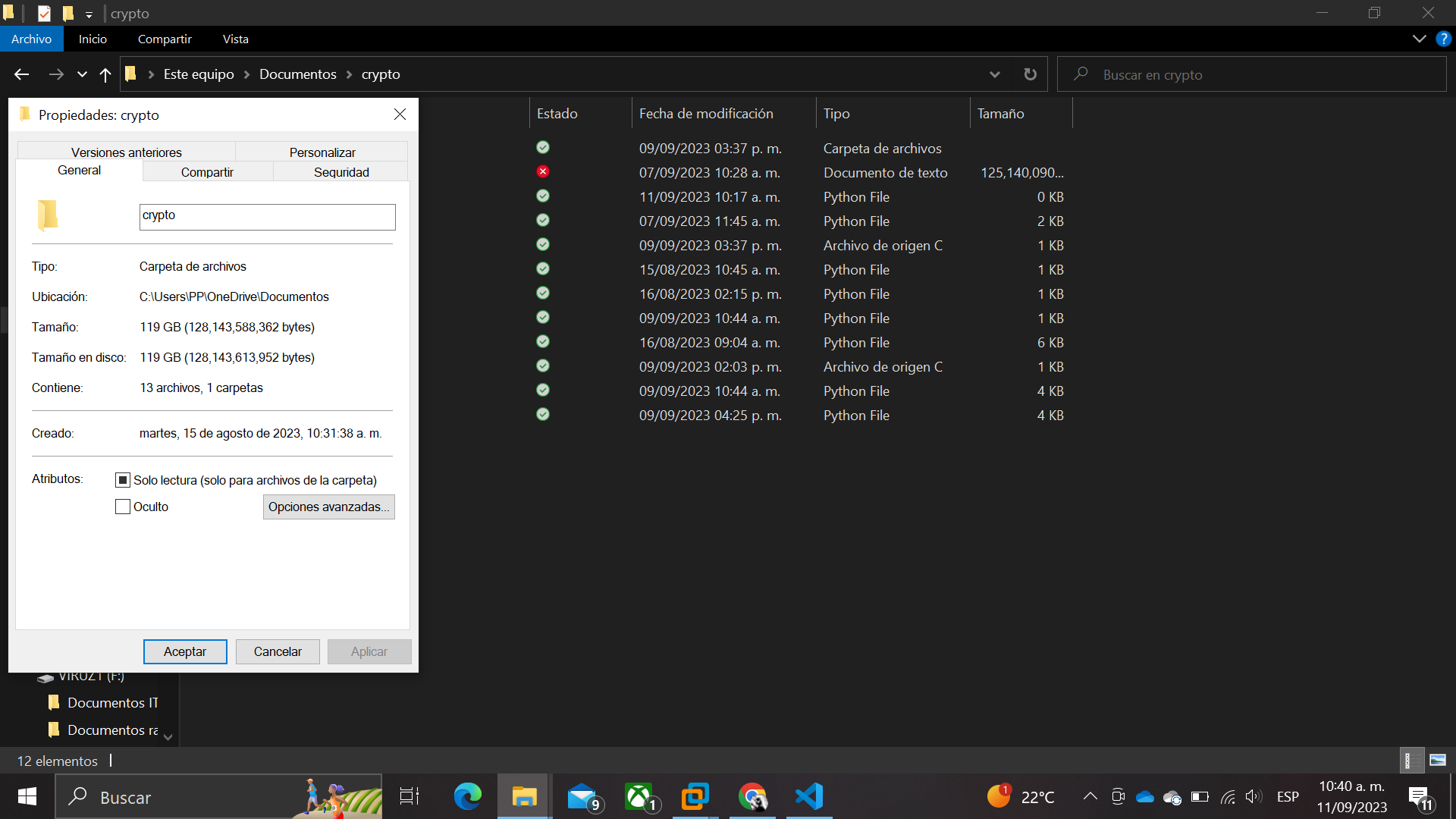
* **Baby steps → Giant steps**

Con este algoritmo, es posible resolver el problema planteado; se llegó a la conclusión de que no es rentable con el hardware que disponemos, puesto que por cada nueva ejecución con nuevos valores, se requerirían 128Gb disponibles, ya fuese de memoria RAM o de almacenamiento físico.

Sin mencionar que, a pesar de contar con una lista Arcoiris previamente generada esta toma una excesiva cantidad de tiempo en iterar. Aun con una eficiencia de , es bastante menos eficiente que otros algoritmos.

Aquí podemos observar la cantidad de espacio que ocupa la lista Arcoiris generada en un archivo de texto, para un número de 64 bits. Podemos observar que el archivo supera los 100Gb de almacenamiento:





### 

### **Referencias extra:**

Conrad, S. (2015, May 22). Logjam attack, similar to the FREAK vulnerability, breaks TLS security. Emsisoft | Cybersecurity Blog. <https://www.emsisoft.com/en/blog/16757/logjam-attack-similar-to-the-freak-vulnerability-breaks-tls-security/#:~:text=The%20Logjam%20attack%20technique%20involves,passing%20through%20the%20affected%20connection>

‌Paar, C., & Pelzl, J. (2009). Understanding Cryptography: A Textbook for students and practitioners. <http://euro.ecom.cmu.edu/resources/elibrary/epay/Sigs.pdf>

[Logjam Vulnerability] Explanation and Prevention Guide. (2021, April 2). <https://crashtest-security.com/logjam-tls/>

Pollard’s Rho Algorithm for Prime Factorization. (2016, January 3). GeeksforGeeks. <https://www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/>

‌

‌