Funciones de Hash La librería PyCryptoDome tiene funciones de hash para varios algoritmos. Vamos a cargar algunas de ellas. La lista completa está en: https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html (Recuerda: MD5 está obsoleto y roto, no se tiene que utilizar en aplicaciones reales) In [1]: !pip install pycryptodome from Crypto.Hash import MD5, SHA256, SHA512, SHA3_256 Requirement already satisfied: pycryptodome in /home/gattes/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages (3.17) [notice] A new release of pip is available: 23.0 -> 23.1 [notice] To update, run: pip install --upgrade pip Tradicionalmente, una función de hash se va alimentanto de datos (update ()) hasta que necesitas el hash (también llamado "digest"). Este procedimiento de update () ofunciona, por ejemplo, si tienes que calcular el hash de un conjunto de mensajes o lees los bytes bloque a bloque de un archivo grande. In [2]: hasher = SHA256.new() hasher.update(b'hola') hasher.update(b'mundo') hash1 = hasher.hexdigest() print(hash1) 93fa3e4624676f2e9aa143911118b4547087e9b6e0b6076f2e1027d7a2da2b0a Si ya conoces el mensaje puedes hacerlo todo en una sola línea como en el ejemplo siguiente. Observa que el hash coincide con el calculado antes. In [3]: hash2 = SHA256.new(data=b'holamundo').hexdigest() print(hash2) print(hash1 == hash2)93fa3e4624676f2e9aa143911118b4547087e9b6e0b6076f2e1027d7a2da2b0a True ¿Qué pasa si cambiamos ligeramente el mensaje? Por ejemplo, añadimos un espacio, o ponemos letras en mayúsculas, o signos de admiración.. In [4]: hash2 = SHA256.new(data=b'hola mundo').hexdigest() print(hash2) print(hash1 == hash2)0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f False Preguntas • Prueba varios cambios en el mensaje "hola mundo" y apunta los hashes, verás que cambian totalmente por muy pequeños que sean los cambios: algunas mayúsculas, números, signos de puntuación... In [7]: ex_hash = SHA256.new(data=b'Hola Mundo').hexdigest() print(f"Hash de 'Hola Mundo': {ex_hash}") ex_hash = SHA256.new(data=b'Hola Mundo!').hexdigest() print(f"Hash de 'Hola Mundo!': {ex_hash}") ex_hash = SHA256.new(data=b'H014 Mund0!').hexdigest() print(f"Hash de 'H014 Mund0!': {ex_hash}") Hash de 'Hola Mundo': c3a4a2e49d91f2177113a9adfcb9ef9af9679dc4557a0a3a4602e1bd39a6f481 Hash de 'Hola Mundo!': d4962daf2b2f39666bcd8d35df1357c5608b7019791c20812cd9108b830388bc Hash de 'H014 Mund0!': b3517fe8786cc6d8ed2afa9b35ef1dddfa2ccabdf5962e19c599fa2cca87d36f ¿Cuántos mensajes existen que tengan el mismo hash que "hola mundo"? **Definicion**: A la situación donde (al menos) 2 mensajes comparten el mismo hash, se la conoce como colision. Respuesta: La respuesta facil es que existen infinitos mensajes, dado que existen infinitos inputs para un algoritmo que tiene un output finito (256 bits en el cash de SHA-256). Como caso particular a SHA-256, dado que este algoritmo produce un hash de 256 bits de largo, las posibilidades de encontrar 2 mensajes con el mismo hash es realmente baja. dado que este algoritmo produce un hash de 256 bits de largo, las posibilidades de encontrar 2 mensajes con el mismo hash es realmente baja. Por otro lado y como hemos visto, un pequeño cambio en el input original altera el output del hash que ni siquiera parece que el nuevo hash derive del mismo dato. A esto se lo conoce en criptografia como efecto avalancha ¿Podrías encontrar alguno de estos mensajes que tengan el mismo hash que "hola mundo"? Respuesta: Virtualmente es imposible, pero suponiendo que lo queramos hacer podria solo a traves de un ataque de fuerza bruta, para este caso necesitaria un computador especializado (ejemplo un AntMiner) para generar hashes y tratar de encontrar dos que sean iguales. Otra opcion es buscar en las conocidas Rainbow Tables. Calcula el valor de hash de un archivo de texto con el texto "hola mundo" en tu ordenador desde línea de comandos. ¿Coincide con el hash anterior? • Powershell en Windows: Get-FileHash NOMBREDEARCHIVO • Linux/OSX: sha256sum NOMBREDEARCHIVO Respuesta: Si, es el mismo hash2 = SHA256.new(data=b'hola mundo').hexdigest() print(hash2) 0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f \$ echo -n "hola mundo" > archivo_de_texto \$ sha256sum archivo_de_texto 0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f archivo_de_texto Cambia el nombre del archivo y calcula su hash. ¿Ha cambiado el hash al cambiar el nombre del archivo? Respuesta: No, no ha cambiado: \$ mv archivo_de_texto archivo_de_texto_de_prueba \$ cat archivo_de_texto_de_prueba hola mundo \$ sha256sum archivo_de_texto_de_prueba 0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f archivo_de_texto_de_prueba Tamaño de un resumen hash Fíjate: el hash SHA256 siempre tiene la misma longitud, sea como sea de largo el texto de la entrada. SHA256: longitud 256 bits • SHA512: longitud 512 bits In [12]: print(SHA256.new(data=b'hola').hexdigest()) print(SHA256.new(data=b'hola mundo').hexdigest()) print(SHA256.new(data=b"""Cryptographic hash functions take arbitrary binary strings as input, and produce a random-like fixed-length output (called digest or hash value). It is practically infeasible to derive the original input data from the digest. In other words, the cryptographic hash function is one-way (pre-image resistance). Given the digest of one message, it is also practically infeasible to find another message (second pre-image) with the same digest (weak collision resistance). Finally, it is infeasible to find two arbitrary messages with the same digest (strong collision resistance). Regardless of the hash algorithm, an n bits long digest is at most as secure as a symmetric encryption algorithm keyed with n/2 bits (birthday attack). Hash functions can be simply used as integrity checks. In combination with a public-key algorithm, you can implement a digital signature.""").hexdigest()) b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79 0b894166d3336435c800bea36ff21b29eaa801a52f584c006c49289a0dcf6e2f 000bdd11b8fe147e274127d1d18edcb9d4acff62c96f7e8543de3b7d90a185c5 En los ejemplos anteriores hemos utilizado la función hexadecimal) y el número 254 (decimal) como 'fe' (hexadecimal). Fíjate: 8 bits son un byte, es decir, un número entre 0 y 255 (en decimal), es decir, un número entre 00 y ff (en hexadecimal). **Un byte son dos caracteres hexadecimales**. Podemos accede a la cadena binariade bytes, sin pasarla a hexadecimal, utilizando la función digest() en vez de hexdigest(). Pero no podríamos imprimirla. Así que: • El resumen SHA256 es de 256 bits, sea como sea el tamaño de la entrada 256 bits son 32 bytes Que se representan como 64 caracteres hexadecimales • Pero ambas representaciones son equivalentes. Simplemente, una podemos imprimirla y la otra no. A veces querremos imprimir hashes y por eso es común que los veamos en hexadecimal In [14]: hasher = SHA256.new(data=b'hola') hash_bytes = hasher.digest() hash_hexa = hasher.hexdigest() print(f'Valor de hash SHA256 en binario. Longitud={len(hash_bytes)} bytes Valor={hash_bytes}') print(f'Valor de hash SHA256 en hexadecimal. Longitud={len(hash_hexa)} caracteres Valor={hash_hexa}') Valor de hash SHA256 en binario. Longitud=32 bytes Valor=b'\xb2!\xd9\xd9\xd9\xb0\x83\xa7\xf34(\xd7\xc2\xa3\xc3\x19\x8a\xe9%aMp!\x0e(g1\xca\xa7\xcdM\xdby' Valor de hash SHA256 en hexadecimal. Longitud=64 caracteres Valor=b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79 Por tradición, se ha preferido codificar los hashes en hexadecimal y no en base64, que hubiese sido otra opción perfectamente válida. Velocidades de cálculo de hash Vamos a calcular cuántos hashes podemos calcular por segundo. OJO: este cálculo simplemente nos vale para comparar algoritmos entre sí. Jupyter no tiene acceso a la GPU de tu ordenador, ni Pycryptodome está pensada para gran velocidad. Si exprimes tu ordenador con programas externos seguro que obtendrás números mucho mayores. La ejecución de estas líneas puede llevar desde varios segundos a un minuto. Fíjate en que el bloque empieza con In[*] mientras está haciendo cálculos. In [15]: **import** timeit NUM=500000 print(f'Calculando {NUM} hashes en MD5, SHA256, SHA512, SHA3_256...') time_md5 = timeit.timeit(lambda: MD5.new(data=b'hola').hexdigest(), number=NUM) time_sha256 = timeit.timeit(lambda: SHA256.new(data=b'hola').hexdigest(), number=NUM) time_sha512 = timeit.timeit(lambda: SHA512.new(data=b'hola').hexdigest(), number=NUM) time_sha3 = timeit.timeit(lambda: SHA3_256.new(data=b'hola').hexdigest(), number=NUM) print(f'MD5: spent={time_md5} s speed={NUM / time_md5:.{2}} H/s') print(f'SHA256: spent={time_sha256} s speed={NUM / time_sha256:.2f} H/s') print(f'SHA512: spent={time_sha512} s speed={NUM / time_sha512:.2f} H/s') print(f'SHA3_256: spent={time_sha3} s speed={NUM / time_sha3:.2f} H/s') Calculando 500000 hashes en MD5, SHA256, SHA512, SHA3_256... MD5: spent=4.759243090000382 s speed=1.1e+05 H/s SHA256: spent=6.087186594999366 s speed=82139.75 H/s SHA512: spent=8.049414494998928 s speed=62116.32 H/s SHA3_256: spent=5.789126294999733 s speed=86368.82 H/s Preguntas ¿Cuál de los algoritmos es más rápido? ¿Cómo afecta doblar el número de bits (es decir, pasar de 256 a 512 bits)? • El mas rapido es MD5, aunque como está obsoleto y roto podriamos decir que SHA3_256 es el mas rapido, con una velocidad de ~86368 Hashes/segundo dado que fue diseñado para ser mas performance. • Con respecto al numero de bits, el doblar la cantidad de bits afecta dependiendo del largo del mensaje, dado que para la familia SHA-2, el proceso es: 1. Padding: Se realiza padding para agregar bits al mensaje original para que sea del mismo largo definido por el estandar de la funcion. La cantidad de bits menos que un multiplo de 512 luego de que se agregan estos bits. Similar a esta funcion: $n \times 512 = M + P + 64$ • n es un numero Natural a fin de alcanzar un multiplo de 512 (bits) • M es el largo (en bits) del mensaje original P es la cantidad de bits del proceso de Padding 2. Agregar bits: Una vez hecho el padding sobre el mensaje original, agregamos los 64 bits restantes para tener un mensaje final multiplo de 512. Dado este proceso podemos inferir el largo de bits va a ser beneficioso, dependiendo la cantidad de datos que vayamos a pasar por nuestro algoritmo de hashing. Por tanto SHA-512 sera mas rapido que SHA-256 a mas datos a hashear tengamos. Esto lo podemos ver mucho mas claro en la siguiente prueba: \$ openssl speed sha256 sha512 Doing sha256 for 3s on 16 size blocks: 17219108 sha256's in 3.00s Doing sha256 for 3s on 64 size blocks: 13006138 sha256's in 3.00s Doing sha256 for 3s on 256 size blocks: 7849050 sha256's in 3.00s Doing sha256 for 3s on 1024 size blocks: 2994971 sha256's in 3.00s Doing sha256 for 3s on 8192 size blocks: 454852 sha256's in 3.00s Doing sha256 for 3s on 16384 size blocks: 229878 sha256's in 3.00s Doing sha512 for 3s on 16 size blocks: 7422390 sha512's in 3.00s Doing sha512 for 3s on 64 size blocks: 7277047 sha512's in 3.00s Doing sha512 for 3s on 256 size blocks: 3478864 sha512's in 3.00s Doing sha512 for 3s on 1024 size blocks: 1334032 sha512's in 3.00s Doing sha512 **for** 3s on 8192 size blocks: 196418 sha512's in 3.00s Doing sha512 for 3s on 16384 size blocks: 99797 sha512's in 3.00s version: 3.0.2 built on: Mon Feb 6 17:57:17 2023 UTC The 'numbers' are in 1000s of bytes per second processed. 16 bytes 64 bytes 256 bytes 1024 bytes 8192 bytes 16384 bytes 91835.24k 277464.28k 669785.60k 1022283.43k 1242049.19k 1255440.38k sha256 39586.08k 155243.67k 296863.06k 455349.59k 536352.09k 545024.68k Vemos que SHA-512 tiene una performance visiblemente mayor a medida que mas datos son hasheados. Calcula el hash SHA-256 y SHA-512 de un archivo de unos 500MB en tu ordenador (por ejemplo, una película) ¿Cuánto tiempo le lleva? Creando el archivo: \$ head -c 500M </dev/urandom > /tmp/test_file \$ du -h test_file 501M test_file SHA256: \$ time sha256sum test_file 19a690aed954d8a67d0cda3062c1672f181bcb09c90dce49703252fab20aaf95 test_file 0m2.714sreal user 0m2.649s0m0.064ssys SHA512: \$ time sha512sum test_file 3dacb8eae1253581e161ea69590b6e37b82efb820117deccefc3ceef3b2df414a077685b92a2455918ecbbbfed7d25516d6001801e7a4701db5509c9091db7e8 test_file real 0m1.769suser 0m1.692s0m0.076s Para SHA-256 el tiempo es de ~2.7 segundos, mientras que para SHA-512 es de ~1.7 segundos Proof of work Hashcat Vamos a programar una "Proof of work" (pow) muy sencilla inspirada en Hashcat, la misma tecnología utilizada por bitcoin: "encuentra un número tal que al juntarlo a un texto y calcular su hash, el hash empiece por un número determinado de ceros". La idea es que no se aceptará un correo electrónico, nuevo bloque bitcoin... lo que sea, si el emisor no incluye una "proof of work" junto con el mensaje. Es decir, que pruebe que ha estado X minutos buscando el contador que resulte en un hash correcto. De esta manera se limita el número de mensajes por minuto que puede emitir una persona. Fíjate que encontrar el contador es un proceso costoso, pero comprobar que el contador es correcto es muy rápido: solo tienes que calcular el hash y mirar si empieza con el número correcto de ceros. Nosotros vamos a utilizar el mensaje "hola" solo para ilustrar. Bitcoin funciona de una manera similar, pero el "mensaje" en bitcoin es un bloque con descripciones de transacciones y sus firmas electrónicas. Una propuesta de función (poco eficiente) sería así: In [17]: def proof_of_work(msg, effort): test = '0' * effort ctr = 0 blk = f'{msg}:{ctr}'.encode() h = SHA256.new(data=blk).hexdigest() while not h[:effort] == test: ctr **+=** 1 blk = f'{msg}:{ctr}'.encode() h = SHA256.new(data=blk).hexdigest()return dict(proof=blk, hash=h) Y vamos a probar con dos esfuerzos: que el hash "MSG:CONTADOR" empiece por solo un cero, o que empiece por dos ceros. In [18]: print(proof_of_work('hola mundo', effort=1)) print(proof_of_work('hola mundo', effort=2)) print(proof_of_work('hola mundo', effort=3)) {'proof': b'hola mundo:7', 'hash': '03b259a65fc9611429bfdfc1674d6bf76cdcabfe30b4173668f8a0a8453ce44a'} {'proof': b'hola mundo:537', 'hash': '00956af98878e1a3b97468a9e2399bcd0bdf00caa41b51d9677470697b629f37'} {'proof': b'hola mundo:604', 'hash': '00080de7165b61664c42c3b37691008ad10dbcdb2dd19aff7f1548eb81a04f6b'} ¿Con qué esfuerzo(=número inicial de ceros en el hash) tu PC necesita (más o menos) 30 segundos para encontrar el contador? Utilizando el siguiente script: import time effort = 0elapsed_time = 0 while elapsed_time < 30:</pre> start = time.time() proof_of_work('hola_mundo', effort=effort) end = time.time() effort += 1 elapsed_time = end - start print(f"Con {effort} ceros iniciales, mi ordenador ha demorado {elapsed_time} segundos") Podemos ver que al menos con 7 ceros llega a los 19 segundos. Luego los tiempos son exponenciales, sobrepasando como minimo los 5 minutos. {'proof': b'hola_mundo:0', 'hash': 'f37ffad0bbd62307183ca18c7e6d391458ac409b4806f50bbafd800c2fd6815e'} Con 1 ceros iniciales, mi ordenador ha demorado 0.0001475811004638672 segundos {'proof': b'hola_mundo:17', 'hash': '01ee278cc22f98282bc09d5b0cb51cfd6e4ffab2e6601c0357f03fdd3a1159bd'} Con 2 ceros iniciales, mi ordenador ha demorado 0.0006647109985351562 segundos {'proof': b'hola_mundo:47', 'hash': '008266e1cced9a68cf39d96fa60d9076ab3d148401d01f80d000d804bf18e188'} Con 3 ceros iniciales, mi ordenador ha demorado 0.0016252994537353516 segundos {'proof': b'hola_mundo:821', 'hash': '000c76b79ca0af6a4d71fb8f791a418ba73dc37a9b982d7384b98f0701b6476c'} Con 4 ceros iniciales, mi ordenador ha demorado 0.016820430755615234 segundos {'proof': b'hola_mundo:27394', 'hash': '00008e41a62480ec3b829e2e76b0f63c80168ecc82ae14b123a52fcecf86dd5d'} Con 5 ceros iniciales, mi ordenador ha demorado 0.2442319393157959 segundos {'proof': b'hola_mundo:434540', 'hash': '0000077f55f8a3dd753c561daaa9b3de49e0e510bc6dd53c84a01d57920e3b9c'} Con 6 ceros iniciales, mi ordenador ha demorado 4.230031967163086 segundos {'proof': b'hola_mundo:1227096', 'hash': '0000001cce37399d7cac8dbb2c316009d93b71d8ed217fc80bfd47e06d5feac1'} Con 7 ceros iniciales, mi ordenador ha demorado 19.68189573287964 segundos Firma Digital Vamos a aprovechar lo que ya sabemos de cifrado asimétrico y hashes para ver cómo funciona una firma digital. Alice firma un documento In [22]: document = b"""Cryptographic hash functions take arbitrary binary strings as input, and produce a random-like fixed-length output (called digest or hash value). It is practically infeasible to derive the original input data from the digest. In other words, the cryptographic hash function is one-way (pre-image resistance). Given the digest of one message, it is also practically infeasible to find another message (second pre-image) with the same digest (weak collision resistance). Finally, it is infeasible to find two arbitrary messages with the same digest (strong collision resistance). Regardless of the hash algorithm, an n bits long digest is at most as secure as a symmetric encryption algorithm keyed with n/2 bits (birthday attack). Hash functions can be simply used as integrity checks. In combination with a public-key algorithm, you can implement a digital signature.""" Vamos a generar un par de claves RSA para Alice: una pública alice_pk y otra privada alice_sk. Recuerda: la clave pública la conoce todo el mundo, la clave privada solo la conoce Alice. Ya veremos cómo se distribuye esa clave pública. NOTA: En un entorno real esto se hace mucho antes de firmar: ¡el par de claves debería estar preparado y la clave pública distribuida desde meses antes de la firma! Veremos esto en el tema de PKI In [23]: # Clave de 2048 bits de Alice, pública y secreta from Crypto.PublicKey import RSA alice_sk = RSA.generate(2048) # Clave secreta de Alice alice_pk = alice_sk.publickey() # Clave pública de Alice PyCryptodome ya incluye un módulo para firmar usando las recomendaciones PKCS1. Vamos a aprovechar el módulo, para aprender buenas costumbres y porque PyCryptodome no nos deja utilizar RSA de forma insegura. Ese módulo de firma: Calcula el hash del documento utilizando el hasher que le pasemos (que será SHA256) • Cifra el hash del documento utilizando la clave privada de Alice Todo lo hace siguiendo las recomendaciones PKCS1 • La firma la codifica en base64, para que podamos verla por pantalla (esto no es necesario en realidad) Finalmente, Alice enviaría en un mensaje el documento y su firma. En realidad, lo más probable es que Alice además cifre el documento utilizando algún tipo de cifrado simétrico como AES para proteger su confidencialidad, pero vamos a obviar esa parte en este ejercicio. NOTA: dado que se necesita la clave privada de Alice para firmar, solamente Alice puede generar esta firma de este documento ya que solamente ella conoce su clave privada. In [24]: **from** Crypto.Signature **import** pkcs1_15 from base64 import b64encode, b64decode hasher = SHA256.new(data=document) signature = pkcs1_15.new(alice_sk).sign(hasher) msg = dict(document=document, signature=b64encode(signature)) print(msg) {'document': b'Cryptographic hash functions take arbitrary binary strings as input,\nand produce a random-like fixed-length output (called digest or hash function is oneway (pre-image resistance).\n\nGiven the digest of one message, it is also practically infeasible to find another\nmessage (second pre-image) with the same digest (weak collision resistance).\n\nFinally, it is infeasible to find two arbitrary messages with the same digest\n(strong collision resistance). e).\n\nRegardless of the hash algorithm, an n bits long digest is at most as secure as a\nsymmetric encryption algorithm keyed with n/2 bits (birthday attack).\n\nHash functions can be simply used as integrity checks. In combination with a\npublic-key algorithm, you can implement a digital signatur e.', 'signature': b'aD2Sw5biak/ZZyGhYp+Wd5F3IonuDewI05ZrbgEKeYL7K+1zGen3ucxCLW/p7IHY+PNnYU0eSYURsPYZitMaKgSSmyCfmsnM3t3Teo2V21tEHi9iNCb8PQG909sD/ZrNjWUfdXy8E6IS6ZuHLuPCrGIbAXvSv0DnsS0iX6/0DTnJA90P0aRwpbpg3PSiIl1KLRbDleBCkweijdVB6iZMJxflhu5z/PeJ1imJBba383czzbp3/FHHe1JswclUYS350f/PnitwcD8Y0i6BhQ506L8aV GfcVluJBP2kVjEvKir8zLG1ry4ksxGmnXbYv0TNEUilA0E8/Lr0f5X88pQz9w=='} Pregunta: ¿Por qué crees que Alice cifra **solamente** el hash del mensaje con RSA, en vez de cifrar directamente **todo** el mensaje con RSA? Respuesta: Porque como hemos visto antes RSA dado que no fue diseñado para esto. Con RSA solo podemos encriptar data tan grande como lo es nuestra key (2048 bits o 256 bytes, en nuestro caso). Por otro lado el hash de SHA-256 es de 256 bits (mismo size que nuestra key). Por ultimo, no tiene sentido encriptar toda todo el mensaje, dado que SHA-256 asegura una muy baja probabilidad de encontrar otro texto con el mismo output hash, con lo cual es confiable encriptar solo el hash. Bob verifica la firma de Alice Bob recibe el mensaje msg, que incluye el documento y la firma de Alice, y ya conoce la clave pública de alice_pk de alguna manera (ver tema PKI) Así que Bob hace el proceso inverso: Calcula el hash SHA256 del documento recibido • Decodifica el base64 y descifra la firma recibida utilizando la clave pública de Alice Todo lo hace siguiendo las recomendaciones PKCS1 Como curiosidad, la librería PyCryptodome lanza un error cuando la firma no es válida, y no hace nada si es correcta. Si la verificación de la firma con la clave pública de Alice es correcta, entonces **Bob sabe que el documento lo ha enviado Alice**, y no puede haberlo enviado nadie más. In [25]: rcv_document = msg['document'] rcv_signature = b64decode(msg['signature']) pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature) print("La firma es válida") La firma es válida ¿Qué pasa si un atacante intercepta el mensaje y cambia el documento? Aquí vemos un ejemplo: el atacante ha interceptado el documento y ha puesto información falsa. ¿Puedes identificar qué parte del documento ha cambiado el atacante? Se puede verificar poniendo ambos mensajes en un archivo y utilizando (por ejemplo) diff: \$ diff texto_original texto_cambiado < symmetric encryption algorithm keyed with n/2 bits (birthday attack).</pre> > symmetric encryption algorithm keyed with n/3 bits (birthday attack). Podemos ver que hay cambiado solo 1 numero (del 2 al 3) Observa que ahora la firma de Alice no verifica y la función lanza un error. In [26]: rcv_document = b"""Cryptographic hash functions take arbitrary binary strings as input, and produce a random-like fixed-length output (called digest or hash value). It is practically infeasible to derive the original input data from the digest. In other words, the cryptographic hash function is one-way (pre-image resistance). Given the digest of one message, it is also practically infeasible to find another message (second pre-image) with the same digest (weak collision resistance). Finally, it is infeasible to find two arbitrary messages with the same digest (strong collision resistance). Regardless of the hash algorithm, an n bits long digest is at most as secure as a symmetric encryption algorithm keyed with n/3 bits (birthday attack). Hash functions can be simply used as integrity checks. In combination with a public-key algorithm, you can implement a digital signature.""" pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature) print("La firma es válida") ______ ValueError Traceback (most recent call last) Cell In[26], line 19 1 rcv_document = b"""Cryptographic hash functions take arbitrary binary strings as input, 2 and produce a random-like fixed-length output (called digest or hash value). (\ldots) 16 Hash functions can be simply used as integrity checks. In combination with a 17 public-key algorithm, you can implement a digital signature.""" ---> 19 pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature) 20 print("La firma es válida") File ~/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages/Crypto/Signature/pkcs1_15.py:137, in PKCS115_SigScheme.verify(self, msg_hash, signature) **131** # Step 4 132 # By comparing the full encodings (as opposed to checking each 133 # of its components one at a time) we avoid attacks to the padding 134 # scheme like Bleichenbacher's (see http://www.mail-archive.com/cryptography@metzdowd.com/msg06537). **135** # **136** if em1 not in possible_em1: -> 137 raise ValueError("Invalid signature") 138 pass ValueError: Invalid signature Podemos gestionar los errores con un try/except e informar al usuario. In [27]: **try**: pkcs1_15.new(alice_pk).verify(SHA256.new(data=rcv_document), rcv_signature) print("La firma es válida") except ValueError: print("La firma NO es válida") La firma NO es válida