Laboratori 2: Hash functions

Criptografia i Seguretat

Curs 2023-2024

Arnau Busquets Domingo – 1553345

David Martí Felip – 1633953

Índex

0. Introducció

En aquest Laboratori farem un conjunt d'exercicis per poder veure de forma simplificada i genèrica la dificultat de trobar col·lisions en les funcions hash.

Una funció hash és un algoritme que pren un missatge com a entrada i retorna una cadena fixa de caràcters. Son útils per poder verificar la integritat de les dades, emmagatzemar contrasenya... Però una característica que tenen i tractarem en aquesta pràctica és que no hi ha hashes infinits. Com ja s'ha dit en la definició de hash, tot hash té una cadena fixa de caràcters, per tant, quan es superin el nombre de cadenes diferents amb aquests conjunt de valors començarà a haver-hi col·lisions.

Els tipus de col·lisions que veurem seran les febles i les fortes. Les febles son aquelles en que es busca dos missatges qualsevols amb el mateix hash. Les fortes, per altre banda, parteixen del hash d'un missatge i s'intenta trobar un altre missatge que tingui un mateix hash a aquest primer.

Usant força bruta, generarem valors aleatoris per així poder fer comparatives entre les col·lisions fortes i febles i veure les diferencies entre elles.

1. Exercici 1

Per aquest exercici s'ha de programar una funció que rebi com a paràmetres un missatge (cadena de caràcters) i una nombre N de bits (del 1 al 128), per acabar retornant els N bits més significatius del hash MD5 del missatge representats en format decimal. En el cas de que no pugui fer el càlcul retornarà None.

El codi de la funció usada per resoldre aquest exercici és la següent:

```python
def uab_md5(message: str, num_bits: int) -> Optional[int]:
    if 1 <= num_bits <= 128:
        hash_hexa = hashlib.md5(message.encode()).hexdigest()
        hash_bin = bin(int(hash_hexa, 16))[2:].zfill(128)
        return int(hash_bin[:num_bits], 2)
    return None
```

Per entendre el codi el desglossem:

- hashlib.md5(message.encode()).hexdigest() : Amb la funció message.encode() es converteix el missatge en bytes perquè la funció hashlib.md5() no rebi un

missatge del tipus str sinó un objecte de tipus bytes i amb el ".hexdigest()" transformem a hexadecimal per així tenir el hash en un str.

- bin(int(hash_hexa, 16))[2:].zfill(128) : Apartir del hash en hexadecimal generat anteriorment es pasa a enter per poder així passar-lo amb la funció bin() a binari. Aquest nombre binari descartem els 2 primers valors perquè son el prefix '0b' que posa aquesta funció i per últim s'afegueix amb el zfill(128) per afeguir 0 com padding en cas de que no arribi a 128 bits.
- int(hash_bin[:num_bits], 2) : Per últim es retorna trunca el hash pel nombre de bits que s'han passat per la funció i es pasa aquest valor a enter que serà el hash MD5 final.

Apart del codi també es pot veure com hi ha una excepció per si el nombre de bits no està entre el 1 i el 128. En el cas de que no ho compleixi retornarà None.

2. Exercici 2

En aquest exercici s'ha de implementar una funció que rebi per paràmetre un missatge (cadena de caràcters) i un nombre N de bits (del 1 al 128) i et retorni una tupla amb un missatge (cadena de caràcters) que tingui el mateix hash MD5 que el missatge d'entrada i el nombre de missatges que ha hagut de provar fins a trobar aquesta col·lisió feble (a partir de un hash trobar un altre missatge que tingui el mateix).

```
def second_preimage(message: str, num_bits: int) -> Optional[Tuple[str, int]]:
    primera_imatge = uab_md5(message, num_bits)
    for i in range(100000000):
        segona_imatge = uab_md5(str(i), num_bits)
        if primera_imatge == segona_imatge:
            return (str(i), i)
    return None
```

Desglossem el codi per poder entendre què és el que fa:

- primera_imatge = uab_md5(message, num_bits) : Guardem la imatge (hash) en una variable així podem després comparar les següents imatges amb aquesta.
- for i in range(100000000) : Es comença un bucle on cada iteració serà la creació de una imatge que es comparà amb la primera. Aquest bucle és de com a màxim 100000000 iteracions perquè creiem que el nombre és prou alt com per poder veure si realment la col·lisió és fàcil o no de que passi. En el cas de que es trobi una imatge aquest bucle acabarà.
- segona_imatge = uab_md5(str(i), num_bits) : Usem altre cop la funció "uab_md5" per generar una segona imatge però en aquest cas el missatge que busquem el

hash és el nombre que sigui l'actual iteració, i d'aquesta manera creem un nou missatge i per tant el hash serà probablement diferent.

- if primera_imatge == segona_imatge: return (message + str(i), i) : Aquesta condició només es compleix en el cas de que es faci una col·lisió entre els hashes dels dos missatge i per tant es retornaria el missatge que ha creat la mateix imatge i el nombre d'iteracions que ha tardat en fer-ho.

3. Exercici 3

Acabem de veure la funció per trobar una col·lisió feble, en aquest exercici veurem la trobar una col·lisió forta (busquem una col·lisió de hashes entre dos missatges qualsevols apartir d'un nombre de bits marcat).

Concretament, cal programar una funció que rebi com a paràmetre el nombre N de bits que haurà de tenir els hashes de la col·lisió i retornarà una tupla amb 3 valors, els dos missatges que formen la col·lisió i el nombre d'iteracions que ha calgut fins a trobar aquesta col·lisió. En el cas de que no la trobi, retornarà None.

```python
def collision(num_bits: int) -> Optional[Tuple[str, str, int]]:
    hashes = {}
    for i in range(100000000):
        message1 = str(i)
        hash1 = uab_md5(message1, num_bits)
        if hash1 in hashes:
            return (message1, hashes[hash1], i)
        hashes[hash1] = message1
    return None
```

Desglossem el codi:

- hashes = {} : Creem un diccionari per poder guardar com a clau el hash i com a valor el missatge. Aquesta estructura de dades ens permet conprovar amb facilitat si un hash consta entre els trobats.
- for i in range(100000000) : D'igual manera que en l'altre tipus de col·lisió, fem 100000000 iteracions com a màxim en cas de que no trobi una col·lisió.
- message1 = str(i) hash1 = uab_md5(message1, num_bits) : Primer transformem el nombre de la iteració a un str per després pasar-lo com a missatge en la funció de generar el hash md5.
- if hash1 in hashes: return message1, hashes[hash1], hash1 : Aquesta condició el que mira és si el hash que acabem de generar ha estat mai vist abans (que es trobi entre les claus del diccionari) i en cas de que es compleixi es retorna

ja com a sortida la tupla dels dos missatges i el nombre de iteracions fins a trobar aquest valor.

- hashes[hash1] = message1 : En el cas que no es compleixi la condició, es guardarà el hash entre les claus del diccionari i el valor el missatge. D'aquesta manera les següents iteracions tindràn en compte els hashes trobats abans.

D'igual manera que l'anterior funció, en el cas de que no es trobi cap col·lisió i acabi el bucle es retornarà None.

4. Exercici 4

a. Part A

Un cop ja hem creat totes les funcions per trobar col·lisions, ara cal fer una comparativa per veure com varien els temps per cada tipus. Per aconseguir això el que farem serà avaluar la funció uab_md5 per les diferents mides entre l'1 i el 24, usant diverses mètriques per veure la comparació.

Aquestes mètriques son el temps i el nombre d'iteracions. El nombre d'iteracions ja ens ve donat per les funcions fetes anteriorment, i el temps el calculem amb la funció perf_counter() de la llibreria "time". Apartir d'aquestes noves mètriques farem dues gràfiques per veure la comparativa entre col·lisions. El codi per implermentar-ho tot és el següent:

```
def exercici_4A():

    resultats = []

    for i in range (1, 25):
        start_time_feble = time.perf_counter()
        _, iteracions_feble =  second_preimage("hello", i)
        elapsed_time_feble = time.perf_counter() - start_time_feble

        start_time_forta = time.perf_counter()
        _, _, iteracions_forta = collision(i)
        elapsed_time_forta = time.perf_counter() - start_time_forta

        resultats.append((i,       elapsed_time_feble,       iteracions_feble,
elapsed_time_forta, iteracions_forta))

    df_results  =  pd.DataFrame(resultats,  columns=["Bits",  "Temps  Col·lisió
Feble",  "Iteracions  Col·lisió  Feble",  "Temps  Col·lisió  Forta",  "Iteracions
Col·lisió Forta"])
```

```
    print(df_results)
##########################################################################
    plt.plot(df_results["Bits"],    df_results["Temps    Col·lisió    Feble"],
label="Col·lisió Feble")
    plt.plot(df_results["Bits"],    df_results["Temps    Col·lisió    Forta"],
label="Col·lisió Forta")
    plt.xlabel("Bits")
    plt.ylabel("Temps")
    plt.title("Temps per acabar l'execució en funció del nombre de bits")
    plt.legend()
    plt.show()

    plt.plot(df_results["Bits"],  df_results["Iteracions  Col·lisió  Feble"],
label="Col·lisió Feble")
    plt.plot(df_results["Bits"],  df_results["Iteracions  Col·lisió  Forta"],
label="Col·lisió Forta")
    plt.xlabel("Bits")
    plt.ylabel("Iteracions")
    plt.title("Iteracions per obtenir la col·lisió en funció del nombre de
bits")
    plt.legend()
    plt.show()

    return df_results
```

En aquest codi s'hi pot veure un punt mitg entre la recolecció de les mètriques i el plot de les gràfiques.

- Mètriques: La primera part consisteix en crear un dataframe per poder guardar per cada bit una fila amb el temps d'execuió de cada col·lisió i el nombre d'iteracions que ha fet. El dataframe després de l'execució queda de la següent manera:

	Bits	Temps Col·lisió Feble	Iteracions Col·lisió Feble	Temps Col·lisió Forta	Iteracions Col·lisió Forta
0	1	0.000069	6	0.000009	1
1	2	0.000037	9	0.000005	1
2	3	0.000021	9	0.000005	1
3	4	0.000170	91	0.000005	1
4	5	0.000204	110	0.000007	2
5	6	0.000205	110	0.000018	8
6	7	0.001125	617	0.000018	8
7	8	0.001090	617	0.000051	24
8	9	0.001141	617	0.000077	40
9	10	0.001104	617	0.000076	40
10	11	0.001100	617	0.000081	41
11	12	0.001583	617	0.000147	41
12	13	0.001186	617	0.000087	41
13	14	0.008976	4851	0.000083	41
14	15	0.009284	4851	0.000084	41
15	16	0.161615	71925	0.000469	211
16	17	0.181120	71925	0.000418	211
17	18	0.534645	227345	0.000440	211
18	19	0.549241	227345	0.001897	676
19	20	3.671307	1445175	0.002299	1172
20	21	12.245930	5139452	0.002294	1172
21	22	11.896524	5139452	0.002517	1172
22	23	14.513741	6102231	0.012906	3460
23	24	24.448534	10914642	0.043861	10760

Figura.1. Taula temps i iteracions de les col·lisions febles i fortes

En la Figura 1 s'hi pot veure com a mesura que augmenta el nombre de bits tots els valors augmenten. També es poden veure com a partir dels 19 bits les col·lisions febles donen valors molt major que la resta degut a com augmenta la seva complexitat per trobar segones imatges.

- Gràfiques: En el codi s'hi pot identificar dues gràfiques:
    - Gràfica de temps: En aquesta s'hi veu com el temps varia depenent del tipus de col·lisió al llarg del nombre de bits. És la següent gràfica:

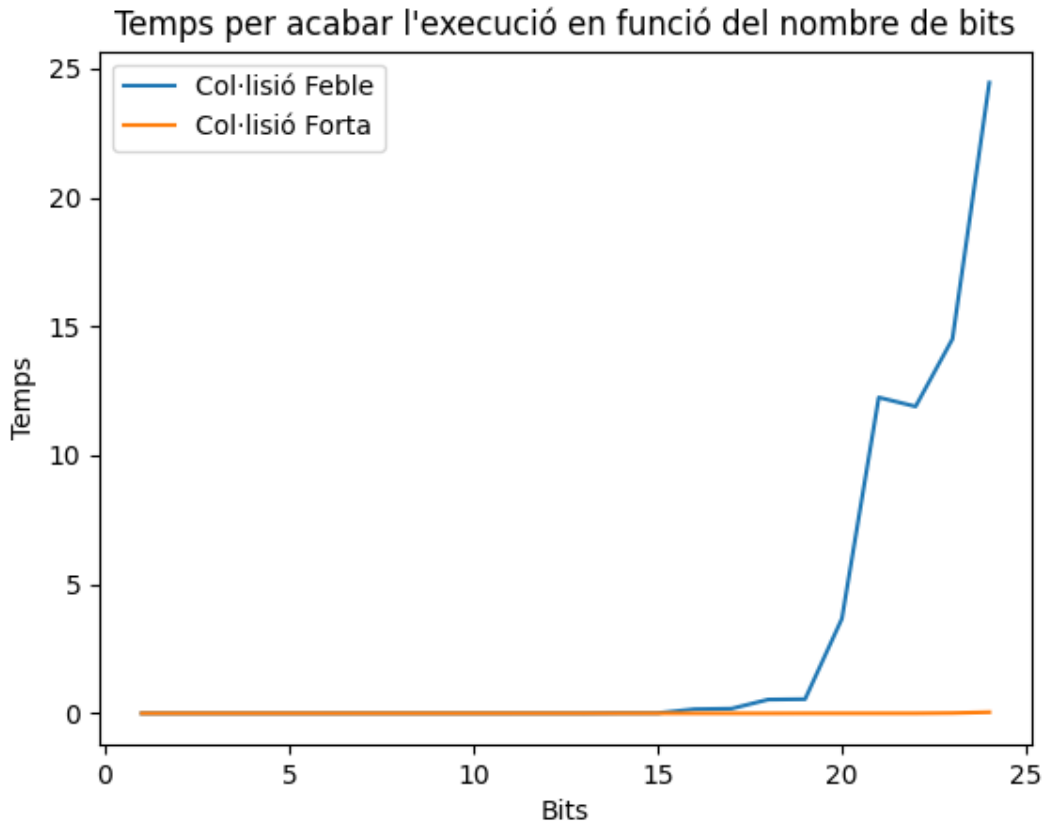

Figura.2. Temps per acabar l'execució en funció del nombre de bits

La diferencia entre la feble i la forta és abismal per aquesta Figura, degut a que la forta es manté en valors propers a 0.0 i en canvi la feble arriba fins a 25 segons, distorsionat la gràfica. Igualment, es pot veure com clarament a mesura que escala el nombre de bits el temps de la forta no varia en gran quantitat, però en canvi, en el cas de les febles sí que ho fa.

- Gràfica de iteracions: En aquesta gràfica es veu com el nombre d'iteracions per trobar la col·lisió varia entre el tipus al llarg del nombre de bits. La gràfica és la següent:

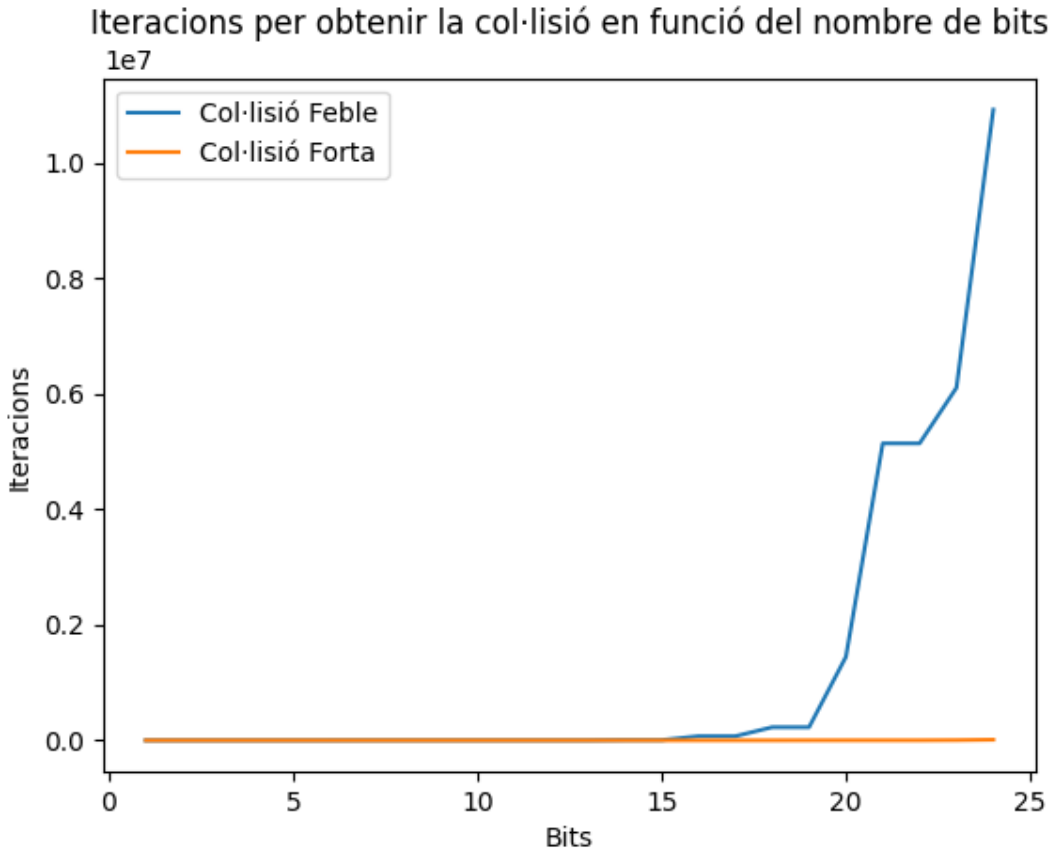

Figura.3. Iteracions per obtenir la col·lisió en funció del nombre de bits

De mateixa manera que en l'anterior, la Figura 3 es veu distorsionada per la gran quantitat d'iteracions que requereix trobar una segona imatge en les últimes iteracions de la col·lisió feble. Clarament es veu com el nombre d'iteracions és molt superior en el cas de les febles que les fortes.

b. Part B

Per últim, volem també veure la diferencia entre el nombre de iteracions teòric amb el real. Per calcular el valor teòric s'ha de fer de diferents maneras per cada col·lisio:

- Col·lisió feble: Com busquem una imatge específica, teòricament, s'han de calcular totes les possibilitats de hash per així trobar com a màxim en l'última iteració. Per tant el valor tèoric és 2^n on n és el nombre de bits.
- Col·lisió forta: En aquest cas, no ens cal tantes iteracions perquè dos hashes qualsevols concideixi, i aquesta afirmació s'explica amb la paradoxa dels aniversaris. La fòrmula per veure quantes iteracions teòriques és $2^{\frac{n}{2}}$ on n és el nombre de bits.

Un cop explicat com s'obtenen els valors veiem el codi per poder fer aquest exercici:

```
def exercici4B(df_results):
```

```
    def iteracions_teoriques_fortes(bits):
        return math.ceil(2 ** (bits / 2))

    def iteracions_teoriques_febles(bits):
        return math.ceil(2 ** bits)

    for i in range(1,25):
        print("feble", i, iteracions_teoriques_febles(i))
        print("forta", i, iteracions_teoriques_fortes(i))

        df_results.loc[df_results.Bits == i, "Iteracions Teòriques Feble"] =
iteracions_teoriques_febles(i)
        df_results.loc[df_results.Bits == i, "Iteracions Teòriques Forta"] =
iteracions_teoriques_fortes(i)

    print(df_results)
##########################################################################
    plt.plot(df_results["Bits"], df_results["Iteracions Col·lisió Feble"],
label="Iteracions Reals")
    plt.plot(df_results["Bits"], df_results["Iteracions Teòriques Feble"],
label="Iteracions Teòriques")
    plt.xlabel("Bits")
    plt.ylabel("Iteracions")
    plt.title("Comparació de iteracions Reals amb Teòriques de les col·lision
Febles")
    plt.legend()
    plt.show()

    plt.plot(df_results["Bits"], df_results["Iteracions Col·lisió Forta"],
label="Iteracions Reals")
    plt.plot(df_results["Bits"], df_results["Iteracions Teòriques Forta"],
label="Iteracions Teòriques")
    plt.xlabel("Bits")
    plt.ylabel("Iteracions")
    plt.title("Comparació de iteracions Reals amb Teòriques de les col·lision
Fortes")
    plt.legend()
    plt.show()

    return df_results
```

En aquest codi s'hi poden veure altre cop dos parts, el càlcul de les iteracions teòriques i les gràfiques.

Les gràfiques son les següents:

- Col·lisió feble:

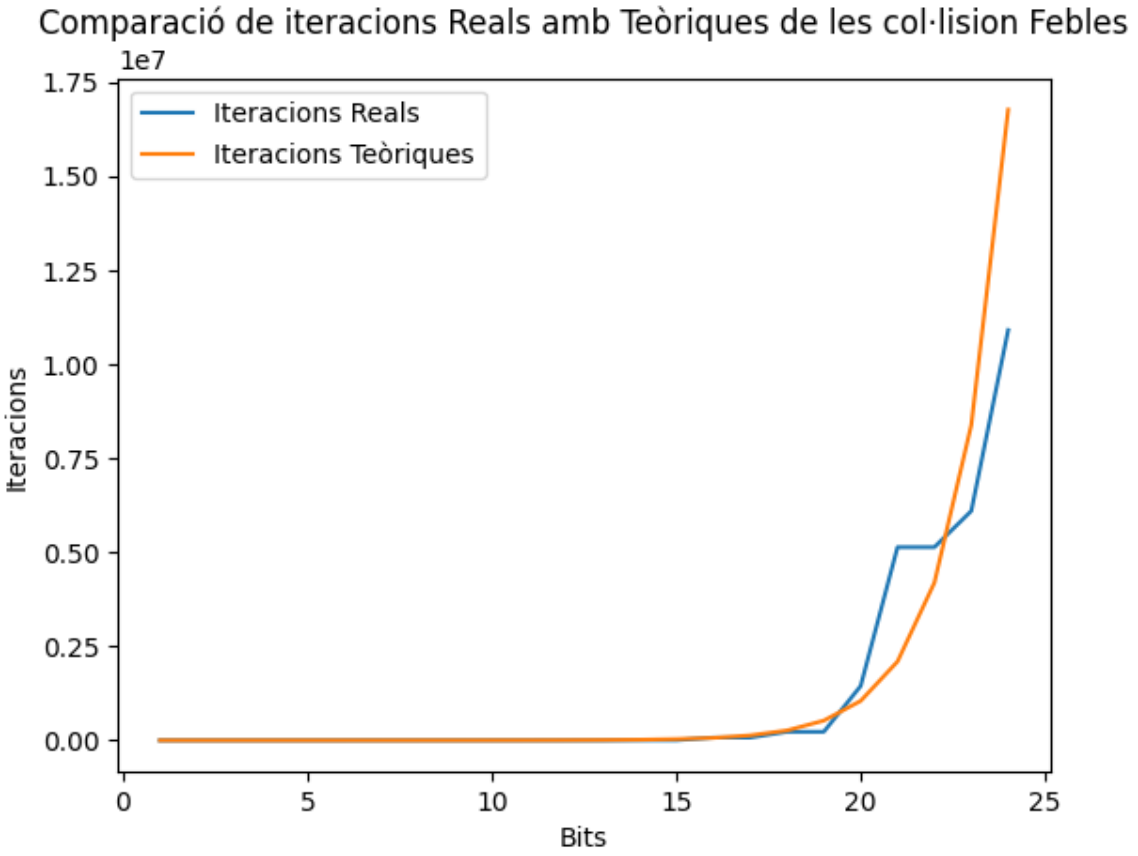

Figura.4. Comparació de iteracions Reals amb Teòriques de les col·lisions Febles

Altre cop, la Figura 4 està distorsionada i per tant les diferencies en els nombres petits de bits no es pot apreciar però no hi ha cap punt diferencia. A partir dels 17 bits sí que comença a haver-hi diferencies i es pot veure com a la iteració 21 hi ha gaires més iteracions de les que teòricament hauria i després passa al reves als 25 bits.

Aquesta poca consistència és deguda a que amb les segones imatges no té perquè que amb 2^n iteracions es trobi resposta ja que pot passar que de totes aquestes imatges es repeteixin imatges ja vistes varis cops així fent que es retrasi aquesta col·lisió.

- Col·lisió forta:

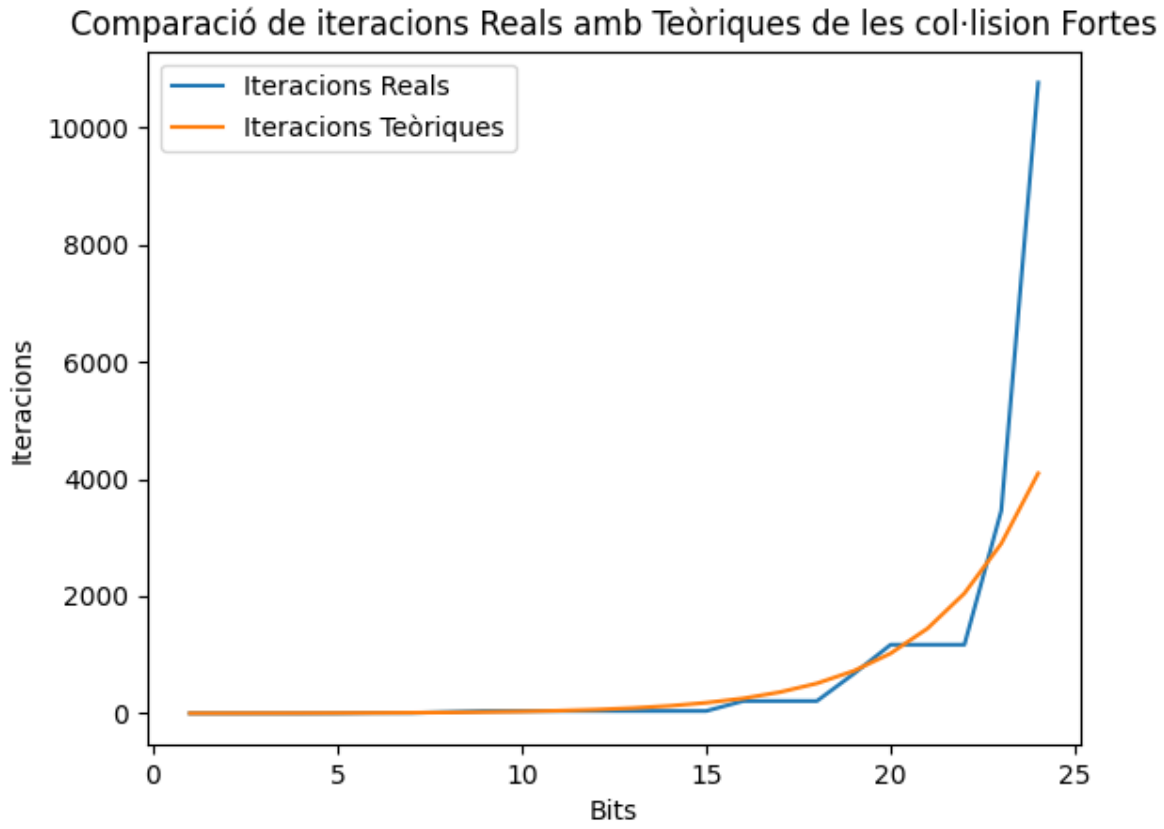

Figura.5. Comparació de iteracions Reals amb Teòriques de les col·lisions Fortes

Com es pot veure en la Figura 5 aquesta gràfica no està del tot distorsionada perquè no té els alts valors que ens trobàvem amb les col·lisions febles. Respecte els valors teòrics i els reals es pot veure clarament com en general tots son molt semblants menys en els 23 i 24 bits on hi ha moltes més iteracions de les esperades.

Aquest pic respecte l'esperat pot ser degut a la mateixa sort de trobar una col·lisió. Aquest valor no pot ser major que $\frac{2^n}{2} + 1$ ja que en el pitjor dels casos, un cop superi la meitat dels possibles hashes ha de trobar una col·lisió de manera obligada.

A diferència de la col·lisió feble, aquesta té un valor màxim.

5. Conlcusions

Per concloure, volem remarcar alguns dels punts que hem trobat més interesant després de fer aquesta Laboratori.

1. La comparació entre les col·lisions: Ha estat el punt que més s'ha tractat al llarg de tota la pràctica i en tota ella s'ha demostrar una clara complexitat major per trobar les col·lisions febles en comparació a les fortes. Tant en el temps i les iteracions ha estat clarament major i no per poc.

2. Implementació de funcions hash: També hem pogut aprendre com s'implementa funcions hash i per a que serveixen per així poder entendre millor aquesta manera de representar els missatges amb una mida fixe de caràcters.
3. Càlcul de les iteracions teòriques amb les reals: Gràcies a un exemple pràctic hem pogut veure com realment els valors teòrics son aproximacions de la realitat ja que al final la probabilística és molt més complexa i encara que sembli que hagi de ser d'una manera hi ha casos en que supera el valor esperat i d'altres que és inferior.