C Labb

Gustav Jenner 20010227 gjenner@kth.se

April 2022

1 Sammanfattning

En egen hashfunktion jämfördes med hashnings algorithmen adler32. Tester utfördes för att undersöka tre olika potentiella svagheter i funktionernas hashning. Vad som undersöktes var funktionernas kollisionsresistans, hur nyckellängden påverkar hashningen samt hur stor "lavineffekt" som funktionerna har. Slutsatsen som kan dras var att den egna hashningen hade en dålig krockresistans då med en lista på 100 platser och 26^3 antal nycklar fylldes endast 11% av listan. Adler32 visade däremot svagheter för småförändringar i nyckeln då förändringar i hashvärdet kunde kopplas till talet 65536. Dessutom visade adler32 en koppling mellan nyckellängd och hashtal-storlek.

2 Uppgiftsbeskrivning

Upgiften går ut på att jämföra två hashfunktioner utifrån ett datasäkerhets perspektiv. De två hashfunktionerna som ska jämföras är en egen skriven funktion och en tilldelad hashfunktion. Funktionerna ska jämföras utifrån tre olika aspekter.

3 Metod

De tre aspekterna av hashfunktionerna som undersöks är kollisionsresistansen, funktionenens "lavineffekt", samt korrelationer mellan nyckellängd och hashlängd.

3.1 Nyckellängds påverkan metod

I en optimal hashfunktion ska inte en kort nyckellängd alltid korrespondera till ett litet hashtal och vise versa. Om en hashning inte har denna säkerhet kommer svagheter finnas hos framförallt kortare nycklar då längden av nyckeln kan knäckas utifrån storleken på hashtalet. Om man vet nyckelängden på ett lösenord är det betydligt snabbare att sedan knäcka nyckeln genom "brute-force". Denna potentiella svaghet undersökts i bägge hashfunktioner genom att plotta hashvärdet för slumpade nycklar mot ökande nyckellängder.

3.2 Kollisionsresistans metod

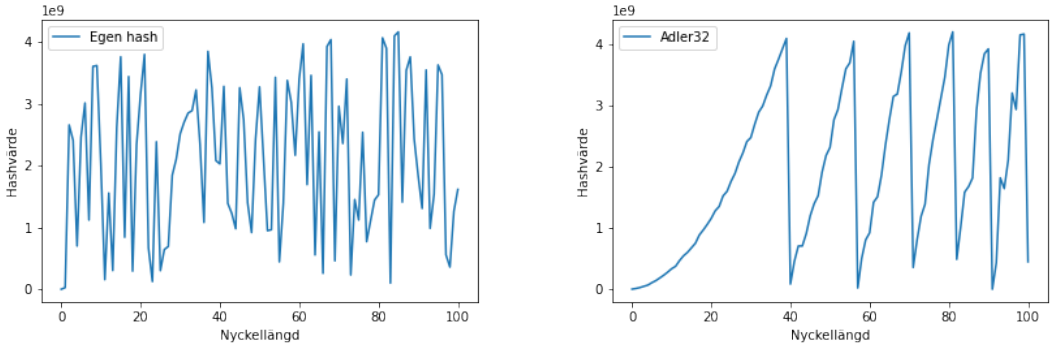
En bred spridning är önskvärt för hashfunktioner från ett datasäkerhets perspektiv. Genom att ha en hashfunktion som genererar unika värden skyddar det mot kollisions-attacker. Dessa attacker kan antingen gå ut på att vilseleda en digital signatur genom att generera olika meddelanden som har samma hashvärde[1]. En annan attack är genom "Hash flooding" som går ut på att skicka stora mängder data som har samma hashvärde till en server så att kollision sker och server får därför långsamma uppslagningar[1]. Kollisionsresistansen testades genom att utnyttja Dirichlets lådprincip[2] för att säkerställa att kollision kommer ske. 26^3 antal hashningar sker genom att hasha alla kombinationer av ett 3 bokstavig sträng baserat på engelska alfabetet, dvs "AAA"- "AAB"...-"ZZZ". Sedan la dessa värden in på dess "hashindex" i en lista med olika längder där krockarna räknades ut för varje index.

3.3 ”Lavineffekt” metod

Lavineffekten innebär att en liten förändring i data innebär en stor förändring i resultat[3], vilket är en viktig egenskap för hashfunktioner. Om nyckeln “A” gav hashvärdet 1 och nyckeln “B” gav hashvärdet 2 är det risk att det går att lista ut nyckeln från hashvärdet samt att hitta liknande hashvärden om man vet nyckeln. ”Lavineffekten” undersöktes genom att beräkna ”distansen” mellan hashvärdena för liknande nycklar till exempel jämför“BBC” med “ABC”, “CBC”, “BAC”, “BCC”, “BBB” och “BBD”. Detta görs för alla nycklarna och medelvärdesskillnaden räknas ut för samtliga nycklar. Detta ger en partiell bild av hur bra hashningarnas “Lavineffekt” är, (se funktionen ”Avalanche_key” i kodavsnittet i appendix för tydligare bild om hur ”distansen” räknas ut).

4 Resultat

4.1 Nyckellängds resultat

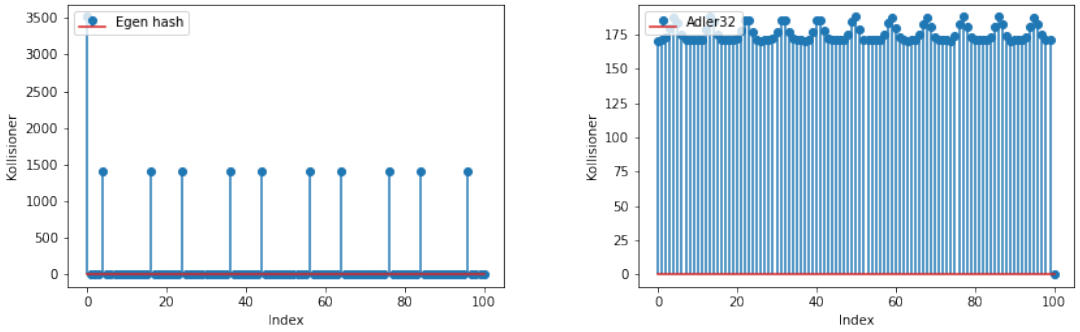


(a) Nyckellängds jämförelse med hashvärde för egen hash

(b) Nyckellängds jämförelse med hashvärde för adler32

Figure 1

4.2 Kollisions resultat



(a) Kollisions test, egen hash för lista med längd 100

(b) Kollisions test, adler32 för lista med längd 100

Figure 2

4.3 Lavineffekt resultat

Nedan är en tabell över den genomsnittliga hashvärde skillnaden för småförändringar i nyckel

Modulo gräns	Egen hash	Adler32
100	37.23	34.14
1000	321.07	369.62
10000	3417.54	3780.37
max gräns*	17621227.67	131073.00

Table 1: Tabell över värdena för Lavineffektsberäkning enligt det som beskrivs för metoden (*Syftar på högsta värdet adler32 kan ge vilket är 4294967295)

5 Analys

5.1 Nyckellängds analys

för den egna hashfunktionen finns det en trend när hashvärdet når modulo gränsen blir hashvärdet efter ofta ett litet tal (se figur 1a). Alltså har grafen en trend av att det är gropar direkt efter toppar. Dock verkar det inte var någon trend i topparna och groparnas frekvens och den egna hashen bör därför inte ha någon tydlig svaghet när det gäller nyckellängden. Detta beror delvis på att algoritmen genererar otroligt stora tal och redan vid nyckellängden 2 nås modulo gränsen (Om nyckeln antas bestå endast av bokstäver).

Däremot har grafen för adler32 ett mycket tydligare beteende (se figur 1b). Hashvärdet ökar parallellt med nyckellängden framtills max hashvärdet nås och sedan börjar cykeln om. Detta faktum skulle gå att utnyttja för att lista ut en viss nyckellängd. Exempelvis om en hemsida har ett lösenords maxlängd på 20 tecken kan man räkna ut den ungefärliga lösenords längden genom att titta på hashvärdet för ett lösenord. Om man får fram att lösenordet har en längd på exempelvis 7 tecken kan sedan lösenordet mycket snabbare tas fram med ”brute force” genom att bara testa kombinationer som är 7 tecken långa.

5.2 Kollisions analys

Den egna hashfunktionen har en stor kollisions risk (se figur 2a). Flest krockar sker vid index 0 runt ≈ 3500 krockar dessutom verkar endast hashfunktionens index finnas vid $20 \times n \pm 4$ där $n = 1, 2, 3, 4$. Den har alltså endast tagit upp 11% av listan. Hashfunktionen har alltså en dålig spriddning och är svag mot tidigare diskuterade ”hash flooding”-attacker. Dock är inte denna svaghet lika stor vid list längder av primtal (se i appendix figur 7) då det blir moduloräkning med primtal.

Adler32 har däremot en bra spridning då för list längden 100 sker det ungefär ≈ 175 kollisioner för samtliga index vilket stämmer överens med $26^3/100$ och tar alltså upp 100% av listan. Adler32 har en markant bättre spriddning för samtliga testade list längder jämfört med den egna hashfunktionen. Slutsatsen för detta test är att trots den förbättrade kollisionsresistansen för längder i primtal för den egna hashningen sker det fortfarande för många krockar. Det gör att den är extremt dålig i framförallt hashtabeller då långa krocklistor kommer att bildas och sökningar i hashtabellen kommer inte vara av O(1). Adler32 har däremot en bra spridning och verkar inte ha några tydliga kollisons svagheter.

5.3 ”Lavineffekt” analys

Ingen påtaglig slutsats kan dras för de tre första tabellvärdena då de skiftar vilken av hashfunktionerna som har störst värde. Däremot visar Adler32 en stor svaghet för kolumnen med maxgräns då $131073 = 65536 * 2 + 1$, där 65536 är konstanten som multipliceras med ena kontrollsumman[4]. Vad detta innebär är att hashvärdet för små förändringar hos en nyckel går att räkna ut för adler32. Exempel nyckeln ”AAA” ger hashvärdet $X = 25755844$ det innebär att nyckeln ”AAB” ger hashvärdet $X + 65536 + 1$ och om det var det andra A:et som istället blev ett B skulle hashvärdet bli $X + 65536 * 2 + 1$ och om det är första A:et som förändras blir det $X + 65536 * 3 + 1$.

6 Appendix

6.1 Kod

Koden som användes förutom globala variablar och kod för att ta fram grafer.

```
#Importerade bibliotek

import random
import zlib
import matplotlib.pyplot as plt
#--------------------
\end{mitned}
\begin{minted}{python}
#De två hashfunktioner som jämfördes

#Den egna hashfunktionen, in parametrar är nyckeln och modulo gränsen
def own_hash(key,space = 4294967295):
    hash_value = 0
    for i, ele in enumerate(key):
        hash_value += 8 ** (i+1) * ord(ele) #tar 8^i*ord(bokstav)
    hash_value *= hash_value #kvadrerar värdet
    return hash_value % space

#Adler32 hashfunktion
def adler32_function(key, space=None):
    s = key.encode("ascii") #göra om till bit tal
    hash_value = zlib.adler32(s)
    if space is None:
        return hash_value
    return hash_value % space #Modulo gräns
#--------------------

#Funktioner för att ta fram alla kombinationer av tre bokstäver

#Funktion som retunerar lista med alla nycklar (26^3)
def All_keys():
    keys_list = []
    for i in range(0, 26):
        for j in range(0, 26):
            for k in range(0, 26):
                key = create_key(i,j,k)
                keys_list.append(key)
    return keys_list

#Funktion som ger nyckel på tre bokstäver där invärdena representerar vilken bokstav
def create_key(first,second,third):
    alphabet = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "
        ,"U", "V", "W", "X", "Y", "Z"]
    return alphabet[first]+alphabet[second]+alphabet[third]
#--------------------

#Funktioner för nyckellängds-test

#Funktion som retunerar lista med hashvärde for olika slumpade nycklar med ökadlängd
def length_test(hash,end_length):
    length_list = []
```

```
    for i in range(0,end_length+1):
        length_list.append(hash(random_string(i))) #lägger till hashvärde för slumpad nyckel med längd
    return length_list

#Funktion som genererar slumpmässig nyckel där inparametern är längden på nyckeln
def random_string(len):
    alphabet = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "
        ,"U", "V", "W", "X", "Y", "Z"]
    string = ""
    for i in range(0,len+1):
        string += alphabet[random.randrange(0,26)]
    return string
#--------------------

#Funktioner för kollisions-test

#Retunerar lista med krockar för olika index
def Collision_test(hash,space):
    collision_list = create_zero_list(space)
    keys_list = All_keys()
    for key in keys_list:
        index = hash(key,space)
        collision_list[index] += 1
    return collision_list

#Retunerar lista med specifierad längd fylld med nollor
def create_zero_list(length):
    zero_list = [0]*length
    zero_list.append(0)
    return zero_list
#--------------------

#Funktioner för lavineffekts-test

#Funktion som retunerar medelvärde distansen för en nyckel
def avalanche_key(hash,key,space):
    ref = hash(key,space) #referens hash
    dist = 0
    key_list = list(key)
    #tar absolut beloppet av skillnaden mellan hashvärdena
    for i in range(0,3):
        y = key_list.copy()
        y[i] = chr(ord(y[i])+1)
        dist += abs(ref-hash(y[0]+y[1]+y[2],space))
        y[i] = chr(ord(y[i])-2)
        dist += abs(ref-hash(y[0]+y[1]+y[2],space))
    return dist/6 #medelvärdet

#Retunerar medelvärde distansen för summan av nycklar
def avalanche_effect(hash,space):
    keys_list = All_keys()
    tot_dist = 0
    for key in keys_list:
        tot_dist += avalanche_key(hash,key,space)
    return tot_dist/(len(keys_list)) #medelvärde
#--------------------
```

6.2 figurer

Ytterligare figurer från kollisons testet

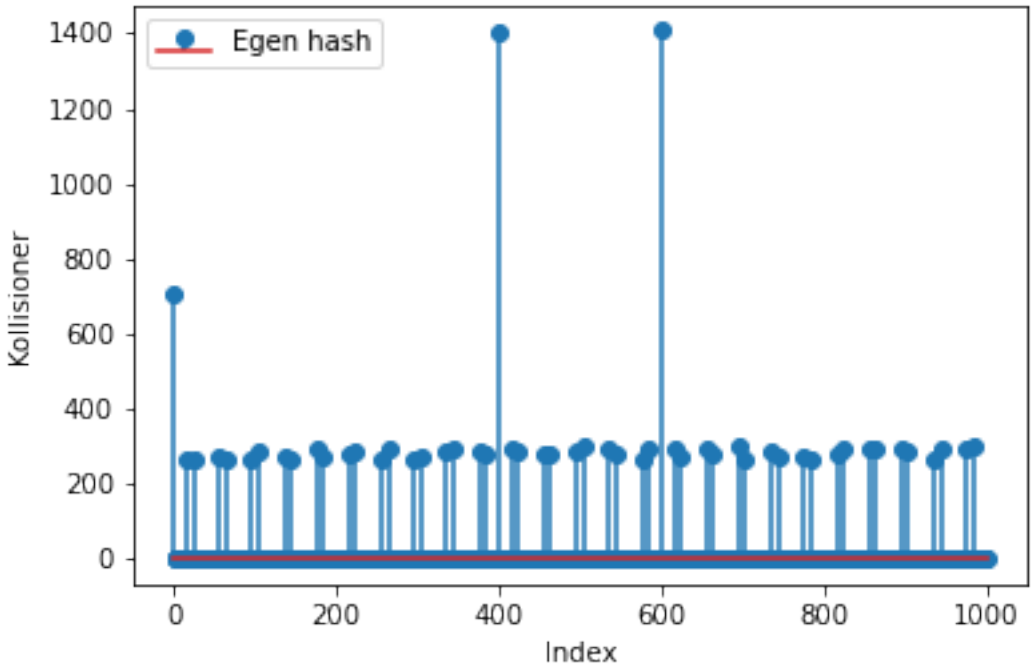


Figure 3: Kollisions test, egen hash för lista med längd 1000

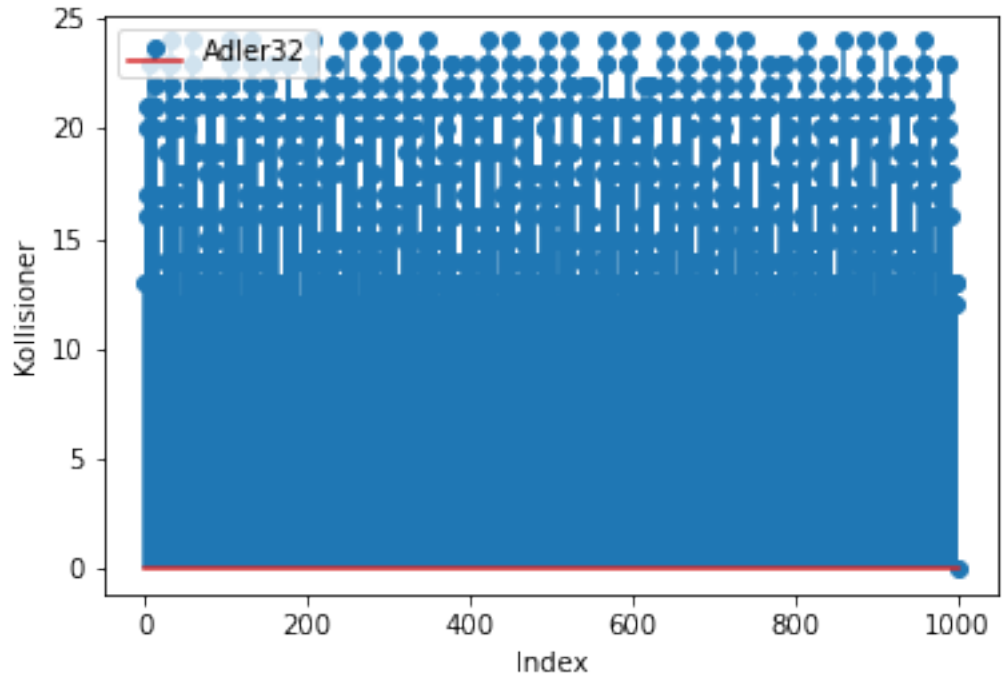


Figure 4: Kollisions test, adler32 för lista med längd 1000

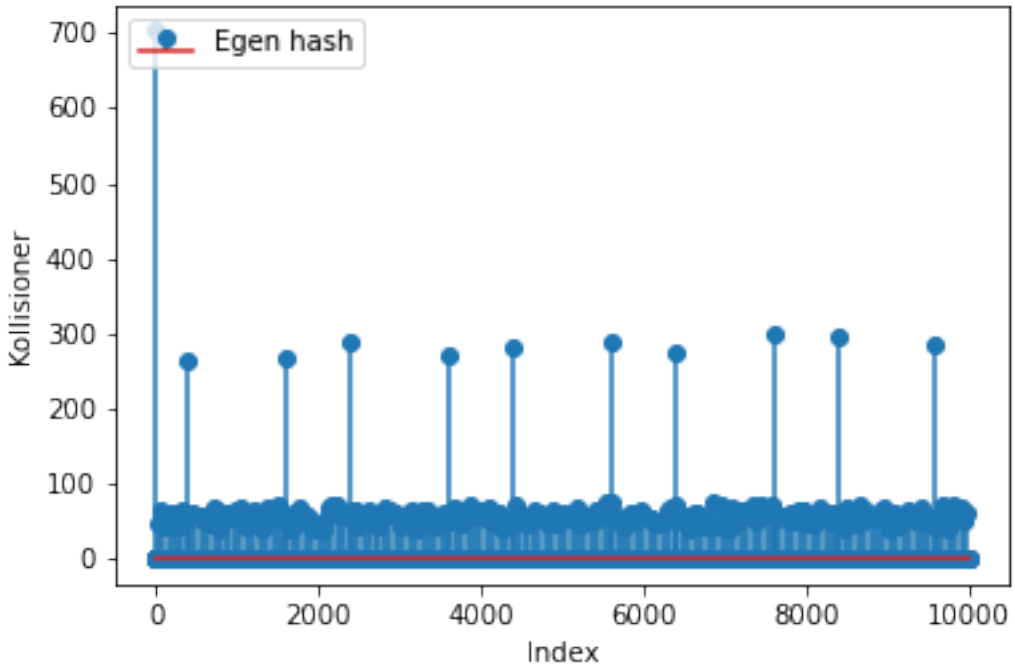


Figure 5: Kollisions test, egen hash för lista med längd 10000

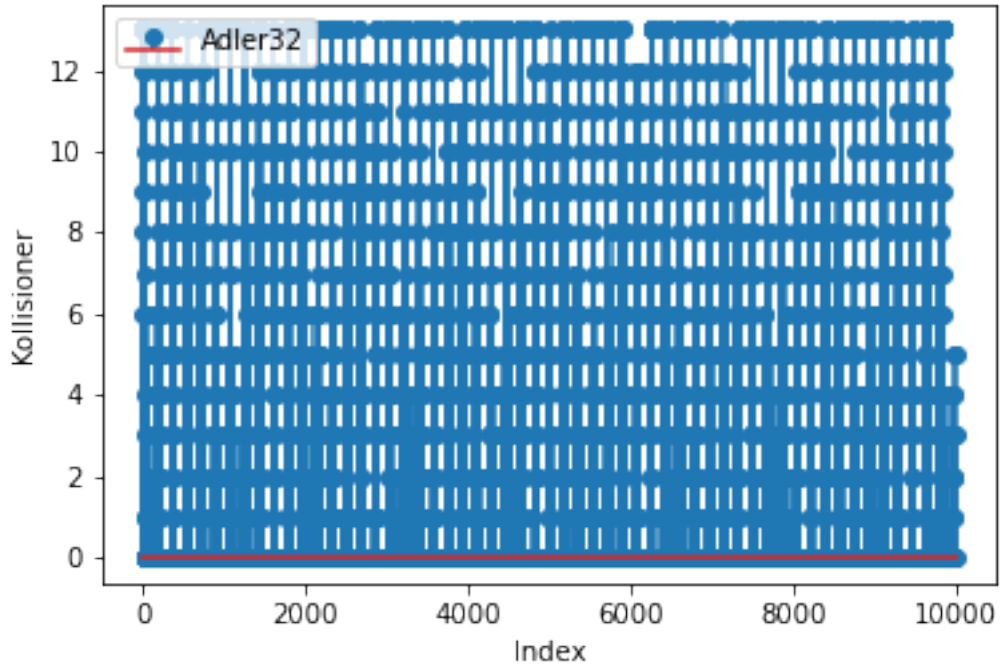


Figure 6: Kollisions test, adler32 för lista med längd 10000

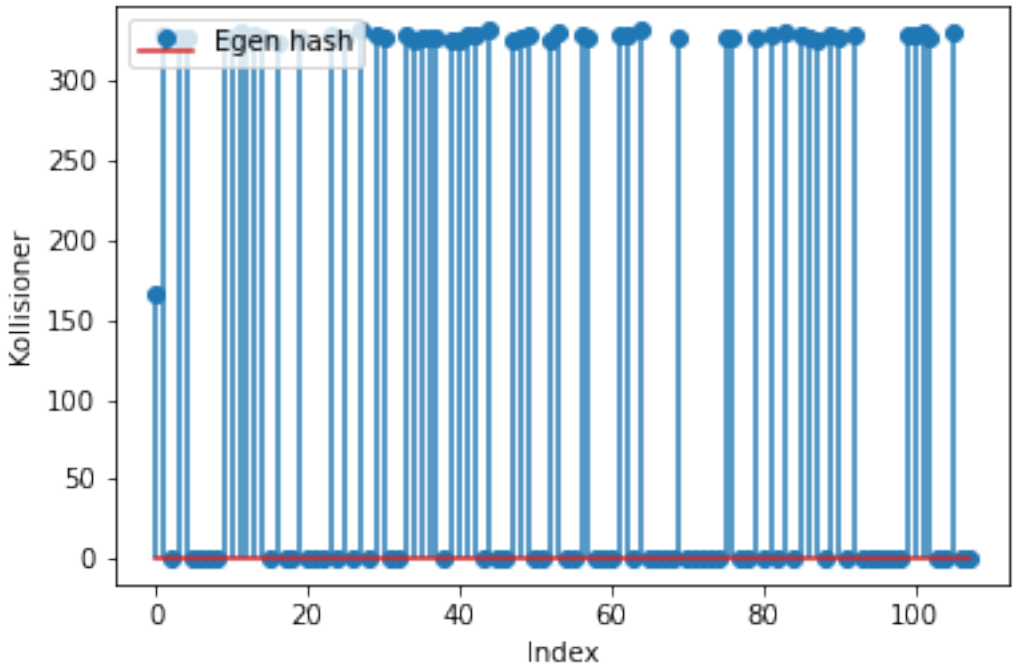


Figure 7: Kollisions test, egen hash för lista med längd 107

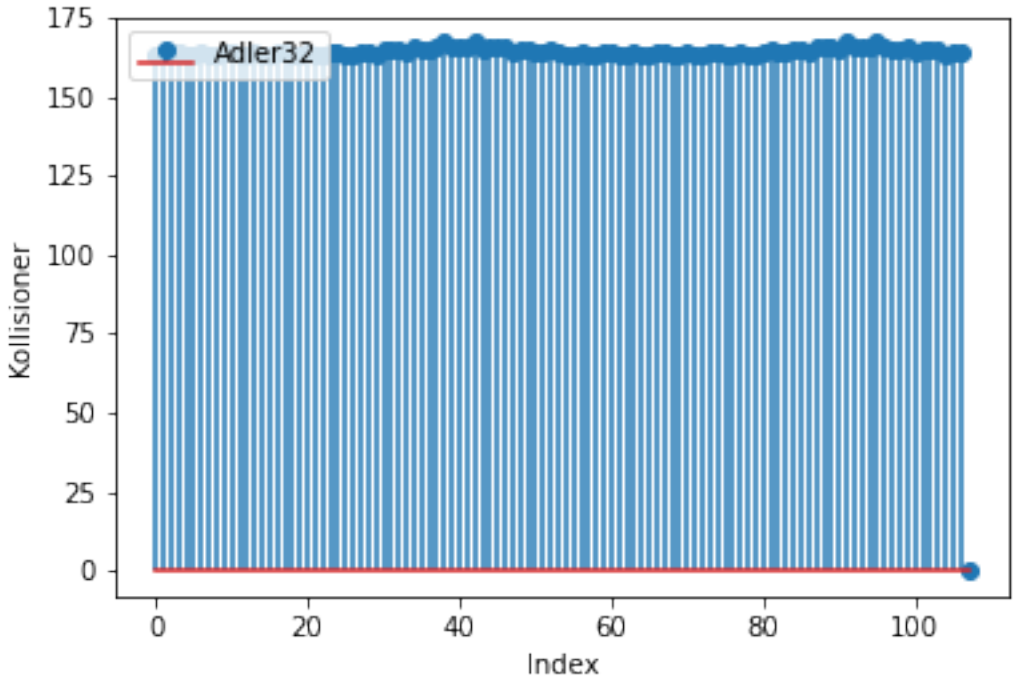


Figure 8: Kollisions test, adler32 för lista med längd 107

7 Källor

[1]"Collision attack - Wikipedia", En.wikipedia.org, 2022. [Online]. Tillgänglig: https://en.wikipedia.org/wiki/Collision_att
[Hämtad: 26- Apr- 2022].
[2]"Pigeonhole principle - Wikipedia", En.wikipedia.org, 2022. [Online]. Tillgänglig: https://en.wikipedia.org/wiki/Pigeonh
[Hämtad: 26- Apr- 2022].
[3]"Avalanche effect - Wikipedia", En.wikipedia.org, 2022. [Online].Tillgänglig: https://en.wikipedia.org/wiki/Avalanche_ef
[Hämtad: 26- Apr- 2022].
[4]"Adler-32 - Wikipedia", En.wikipedia.org, 2022. [Online]. Tillgänglig: https://en.wikipedia.org/wiki/Adler-
32. [Hämtad: 26- Apr- 2022].