Hash Collisions and Birthday Attack

Max Jackson

1 Introduction

We discuss the birthday attack and supplement the results derived with a python program to illustrate the ideas. The birthday attack is centred on the ability to find collisions in the outputs of hash function. We define the necessary concepts below.

A hash function is a computationally efficient algorithm for mapping binary strings to a fixed length binary output. Hash functions enjoy five basic properties:

- The input, x, can be of any length
- The output, h, has a fixed length
- $H(x)$ is easy to compute for any input x
- $H(x)$ is one way. That is to say, given a hash value, h it is computationally infeasible to find an input x, such that $H(x) = h$
- $H(x)$ is collision free

A weakly collision-free hash function is one such that for a given input x it is computationally infeasible to find an input y, not equal to x, such that $H(x) = H(y)$

A strongly collision-free hash function is one for which it is computationally infeasible to find any two inputs x and y such that $H(x) = H(y)$

Many sources which attempt to explain birthday attacks get caught up in the mathematics and relation to the well-known birthday problem that they often fail to contextualise the attack by providing an example of such. We do that now. A typical example of a birthday attack is on digital signatures. Suppose we prepare a legitimate contract m and a fraudulent contract m'. If we can produce enough subtle variations of m and m' such that they retain their original meaning and will not raise suspicion to the recipient (e.g. by containing arbitrary strings), then (with some luck!) we may find a variation of m and m' with $H(m) = H(m')$. In which case, we can get the recipient to sign the legitimate contract m, and attach the signature to m'. This signature also 'proves' the recipient signed the fraudulent contract.

2 Mathematics

Suppose we have a hash function $h(\cdot)$ which yields one of H different values with equal probability, where H is sufficiently large. Now suppose we select n different outputs uniformly at random. Denote by $p(n, H)$ the probability that at least one value is chosen more than once. Then,

$$p(n, H) = 1 - p(\text{All } n \text{ values are unique})$$

Let $p(\text{All } n \text{ values are unique}) = p$, and note that

$$p = \frac{H}{H} \cdot \frac{H-1}{H} \cdots \frac{H-(n-1)}{H}$$

since we have H choice for the first output, then $H - 1$ choices for the second output until we have $H - (n - 1)$ choices for the final output, since at each step the range of accepted output values reduces by one. This condenses to

$$p = \frac{H!}{H^n (H-n)!}$$

taking the logarithm gives

$$\log(p) = \sum_{i=1}^{H} \log(i) - \sum_{i=1}^{n} \log(H) - \sum_{i=1}^{H-n} \log(i)$$

$$= \sum_{i=H-n+1}^{H} \log(i) - \sum_{i=1}^{n} \log(H)$$

re-indexing the first sum gives

$$= \sum_{i=1}^{n} \log(i + H - n) - \sum_{i=1}^{n} \log(H)$$

$$= \sum_{i=1}^{n} \log(1 - \frac{n-i}{H})$$

Since H is sufficiently large, we may employ the Taylor series $\log(1 - x) \approx -x$ so

$$\log(p) \approx -\sum_{i=1}^{n} \frac{n-i}{H}$$

which using standard series gives

$$\log(p) \approx -\frac{n^2 - n(n+1)/2}{H} = -\frac{n(n-1)}{2H} \approx -\frac{n^2}{2H}$$

so upon exponentiating we get

$$p(n, H) \approx 1 - \exp(-\frac{n^2}{2H})$$

Now we ask the question, what is the smallest n such that we have at least probability p of obtaining a collision? Well the above answers the question what is the probability of a collision given n outputs, thus we simply invert the above expression. So denoting this as $n(p, H)$ we see

$$n(p, H) \approx \sqrt{2H \log(\frac{1}{1-p})}$$

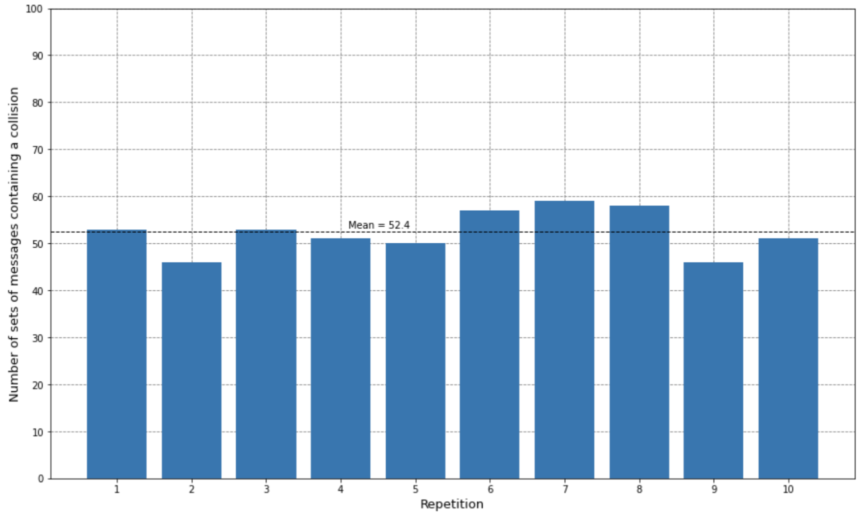
3 Python Implementation

We'll now use python to illuminate the mathematics. First we need some values for H and a probability p. A nice value of p to work is simply $\frac{1}{2}$, and for computational efficiency purposes we'll work with a hash function with an $8-$bit output, yielding $H = 256$.

Using the these values with the final result of the previous section, we can say that roughly half of the time, the digest of $\sqrt{2 \cdot 256 \log(\frac{1}{1-1/2})} \approx 19$ messages should yield a collision.

The code employs the following steps:

- Define a hash function with an 8-bit output.
- Generate 19 messages and check for a collision.
- Repeat this 100 times.
- Count how many sets of digests contain a collision.
- Repeat the 3 previous steps n times to check for consistency of results
- Calculate the mean of the n counts and plot the counts.



Above is the result of running the code with the parameters being 100 sets of 19 messages, repeated 10 times. As expected, roughly 50 out of each 100 sets of 19 messages produce a collision! Below is the code included for interest.

```python
# import libraries
from random import choice, shuffle
import string
import matplotlib.pyplot as plt
import numpy as np

# implement the Pearson hashing function
example_table = list(range(0, 2**8))
shuffle(example_table)

def hash8(message: str, table) -> int:
    hash = len(message) % 2**8
    for i in message:
        hash = table[hash ^ ord(i)]
    return hash
```

The above implements the 8-bit Pearson hash function and the below implements the steps outlined previously.

```python
# define the set of character from which to derive the messages
chars = (string.ascii_letters + string.digits + string.punctuation)

# we initialise the message to a single randomly chosen character
message = choice(chars)

# initialise a list to store the counts
store = []
# define how many times we want to repeat the process
reps = 10

#define how many sets of 19 messages we want to produce at each iteration
sets = 100

# independently repeat the process 'reps' number of times
for _ in range(reps):

    # initialise a counter to track how many sets of messages produce at least one collision
    collisions = 0

    for _ in range(sets):
        # list to store the message digests
        hashes = []

        # generate 19 messages in total. We do this by appending a random character to the message
        # string at each step. The diffusion property of hash functions ensures that the digests
        # are essentially unrelated despite the messages clearly being unrelated, so this shouldn't
        # affect the results by increasing/decreasing the likelihood of a collision between two message
        # digests

        for _ in range(19):
            message += choice(chars)
            hashes.append(hash8(message, example_table))

        # check if we have any collisions and increment the collisions counter by 1 if we do
        collisions += (len(hashes) == len(set(hashes)))

    # append the number of sets of messages that resulted in a collision to the store list
    store.append(collisions)
```

```python
fig, ax = plt.subplots(figsize = (15,9))

ax.set_axisbelow(True)
ax.grid(color = 'gray', linestyle = 'dashed')

ax.bar(list(range(reps)), store)

ax.set_xlabel('Repetition', size = 13)
ax.set_ylabel('Number of sets of messages containing a collision', size = 13)

ax.set_ylim(0, 100)

ax.set_xticks(list(range(reps)))
ax.set_xticklabels(list(range(1,reps+1)))
ax.set_yticks(list(range(0, sets + 1, 10)))

mean = np.mean(store)
ax.axhline(mean, color = 'black', linewidth = 1, linestyle = 'dashed')
ax.text(3.1, mean + 0.8, 'Mean = ' + str(mean))
```

The above is the code used to produce the plot at the end of the last page.