How birthdays can break encryption

An exploration of the birthday paradox and cryptographic hashing algorithms

By Noah Kurrack

Math SL Exploration

11 November 2018

Introduction

Computers are everywhere. In today's society, computers play a vital role in our daily lives. We choose to store a tremendous amount of important information on these electronic devices, from financial records to our personal identity. Assuredly, we take extensive measures to ensure that such sensitive information is properly protected. This is due to the unfortunate consequences of our information falling into the hands of the wrong people, which can be devastating. But what if I told you that birthdays and a little bit of math could jeopardize the security of our private information, no matter how strong your password is?

The issue that I am referring to is known as the birthday problem, and it affects the security of cryptographic hashing algorithms. This is an important issue that I think should be investigated further due to the harmful implications of breaking online security. Additionally, I am interested in computer science which is why I chose to explore math that pertains to computer algorithms. In this exploration, I will explore the math behind the birthday paradox, and how it can be applied to cryptographic hashing. Specifically, I will be discussing the model that is used to find the probability in the birthday paradox and then using that model to predict the probability of a hashing collision. I will compare the predicted probability of a collision to the experimental probability found using a computer program to run the algorithm repeatedly.

What is hashing?

Hashing algorithms are a way of processing and storing information securely. Such an algorithm takes an input of any amount of data and outputs a value that represents the input

(Persits Software, Inc.). From now on, I will refer to the result (or output) of the algorithm as the *hash*. Here are a few examples using the CRC32 algorithm:

Input	Hash
“password”	35C246D5
“How birthdays can break encryption”	31EEA0A8

As you can see, the result has no resemblance to the input data; yet, the result is a computational representation of the data. In other words, I think it makes sense to think of hashing algorithms as a way of summarizing data since the same input will always result in the same hash. The most important property of these algorithms is that they are *irreversible*, hence their popular description—“one way functions” (Persits Software, Inc.).

Due to the irreversible nature of cryptographic hashing algorithms, they are often used to store sensitive information such as passwords. If an unauthorized individual were to gain access to the hashed information, there would not be a way to directly compute the data that produced the hashes (reverse the algorithm). In the same way, hashes are also used to ensure that files are not compromised (e.g contain a virus). The same algorithm is executed with a file as the input, instead of a password. If a file were to contain a virus, the hash of the compromised file would be different than the hash of the original (non-compromised) file.

There is, however, a potential way to negate the security of the hashing algorithm using the birthday paradox. A hacker would not have to figure out your password; instead, he would need to find a collision for the hash of your password. Or, the hacker would have to create a

compromised file that produces the same hash as the original. It is much easier than you might think to find a collision.

Mathematics

Birthday Paradox

The birthday paradox is a well-known probability contradiction arising from the probability of people's birthdays. The situation begins fairly ordinarily. There is a group of 23 people in a room. What is the probability that two of the people share the same birthday? Your first thought is probably that the probability is very low. It seems highly unlikely that any of 23 people could share a birthday when there are 365 possibilities (ignoring leap years for simplicity's sake). Yet, the actual probability is approximately 50%. The only way to fully wrap your head around this reality is to look at the math step by step. In the rest of this section, I will explain the commonly used solution that is outlined in the Wikipedia article "Birthday Problem."

Our goal is to find the probability that two people share the same birthday in the group. We will define this as $P(shared)$; however, it is much easier to find the complementary event, $P(shared')$, which is the probability that no people share the same birthday (we will call the complement $P(unique)$). In general terms, the complement of an event is the probability that the original event does not occur. Since we know that there are 23 people in the group, the basic principle is to find the probability that each person does not share a birthday with anyone else.

Looking at the people one at a time, the probability that the first person shares a birthday with someone else is 0 because we are not yet considering anyone else ($P(same) = 0$;

$P(unique) = 1$). There is not a second birthday to compare the first to. The probability of the second person sharing a birthday with the first person is $\frac{1}{365}$ because the second person's birthday can be any one of the 365 possibilities (again, ignoring February 29th), and only one is the birthday of the first person. The complement of that probability, i.e. the probability that the second person does not share the same birthday, can be found as follows: $1 - \frac{1}{365}$, which equals $\frac{364}{365}$. Now we can multiply the probability of each person having a unique birthday because the events are independent. In other words, the second person's birthday is not affected by what the first person's birthday is. Hence, $P(unique) = \frac{365}{365} \times \frac{364}{365}$. Considering the third person, the probability that this third person does not have either of the first two people's birthdays is $\frac{2}{365}$ with a resulting complement of $\frac{363}{365}$. So $P(unique) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365}$. This pattern continues for each of the 23 individuals in the group. Thus, the probability that no one in the group shares the same birthday can be found as follows: $P(unique) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{343}{365}$ ("Birthday Problem."). Using simple multiplication, $P(unique) \approx 0.49270$. Because this is the complement of the probability we are trying to find, $P(same) = 1 - 0.49270 = 0.5073$, or 50.7%. We have now found the probability of two people having the same birthday in our original group of 23.

We can simplify the above process using pi notation: $P(same) = 1 - \prod_{x=0}^{22}\left(\frac{365-x}{365}\right)$. This notation signifies that we multiply the value in the parentheses for each value of x, 0 through 22 (for a total of 23 values). Pi notation is similar to sigma notation, except you find the product of the values instead of the sum. This notation is adapted from an argument made by Paul Halmos ("Birthday Problem.").

Application to hashing

By now you must be wondering, what does any of this have to do with cryptographic hashing algorithms? Let me explain. It turns out that the same statistical model that we just discussed can be applied to the generation of hashes. Think of it this way. The people in the birthday scenario are now passwords being put into a hashing algorithm, and the birthdays are the possible hashes that the algorithm may produce. This means that we can calculate the probability that two different inputs will generate the same hash. The only difference? The numbers are much bigger.

Let's take the CRC32 hashing algorithm which has 2^{32} different possible hashes. This is because there are 16 possible digits multiplied by 8 digits as described by a technology forum user (Hashem). $16^8 = 2^{32}$. We will also take an arbitrary number of inputs: 80,000. Using the same math that is used in the birthday paradox, we can calculate the probability that two of these inputs will output the same hash (which is called a collision). Thus,

$P(collision) = 1 - \prod_{x=0}^{79999}\left(\frac{2^{32}-x}{2^{32}}\right)$. This probability is equal to 0.525291 or 52.5%. So maybe

80,000 was not quite so arbitrary since the resulting probability is near 50%. We will get to how I came up with that number; let me first explain why it is important to be able to derive it.

If we can easily calculate the probability of a collision using the previously mentioned model, the question now becomes how many inputs it would take to have a 50% probability of producing a collision. In other words, we are solving for the number of inputs that it would take to have a 50% probability of producing a repeated hash, instead of solving for the probability like in the original birthday paradox situation. To extend the original analogy, we are trying to find the number of people we must have in the room to have a 50% chance of two people sharing a birthday. To solve this, we have two options: additional mathematics, or a way to put the model to the test (i.e. a small experiment of sorts). I will show you both so that we can compare the two.

Approximation

Because we do not know the number of inputs that give a certain probability of generating a collision, there is not a way to reverse the model used to find the probability. Therefore, we must use an approximation for the model that can be reversed.

The approximation that we will use is $P(collision) \approx 1 - e^{\frac{-n^2}{2(2^{32})}}$, where n is the number of inputs ("Preshing"). Using a calculator, we can see that this approximation is appropriate for our use (accurate to five decimal places): $1 - e^{\frac{-n^2}{2(2^{32})}} \approx 0.52529$ $(1 - \prod_{x=0}^{79999}\left(\frac{2^{32} - x}{2^{32}}\right) \approx 0.52529)$. This approximation is based on a "first-order Taylor approximation for e^x" which states that $e^x \approx 1 + x$ (Azad). The Taylor series is a bit outside of the scope of this paper and this course so

we will use this approximation without further discussion. For more details on the Taylor Series, see the article cited in my bibliography (Pierce).

The purpose of using this approximation involving e raised to a power is its reversibility —the natural log. This means that $y = e^x$ can be rewritten as $\ln(y) = x$. Therefore, we can rewrite the approximation for the probability of a collision to give us a formula for the number of inputs given a certain probability. This formula is $n = \sqrt{2(2^{32}) \times \ln\left(\frac{1}{1 - P(collision)}\right)}$ ("Birthday Problem."; Azad). Plugging in the probability that we found with the approximated model, we get $n = 80000$ as anticipated.

Experiment

The second method of determining the number of inputs needed to find a collision is actually attempting to find collisions. Using a computer program that I wrote, I decided to find out how many hashes it would actually take to find a collision. I used the previous formula to hypothesize that the average number of hashes need to produce a collision would be approximately 77,163. $\sqrt{2(2^{32}) \times \ln\left(\frac{1}{1 - 0.5}\right)} = 77{,}163$.

Computer program

The computer program that I wrote is relatively simple in design. The basic outline is as follows:

1. Generate a random string of number and letters
2. Compute the resulting hash.

3. Check to see if the hash has been computed previously. If not, store the hash and repeat starting at step 1; if it has, a collision has occurred, so continue to step 4.
4. Output information about collision including hash, both inputs, and the number of hashes computed to find the collision; this information is then recorded in a spreadsheet.

Note: Each iteration of this program is considered one trial. For more details, I have attached a simplified version of the program in Appendix A.

It is important that I discuss a few design decisions that were made in the creation of the program so that my results have credibility. The first decision that I made was to use the CRC32 hashing algorithm. The main reason that I chose this algorithm was its 32-bit hash length, which constitutes 8 characters. This limits the number of potential hashes (called hash space) to 2^{32} and allows a collision to be found using the computational power of a laptop in under a few minutes. An algorithm with a longer hash length would have exponentially more potential hashes, and it would take an unrealistic amount of time to find a collision due to the need for significant computational resources. While the CRC32 hashing algorithm functions similarly to other commonly used hashing algorithms with larger hash sizes, it is not actually used in the current technology industry. CRC32 is a limited implementation of larger algorithms that allows us to perform the computations and gather data more easily.

The other important design factor within the program in the generation of random inputs. If not implemented properly, there could be a risk of using an identical input which would result in the same hash, but not actually be a collision. To mitigate this risk, the number of possibilities for randomly generated inputs must be large enough that the probability of generating the same random input is extremely low. This is really just another use of the birthday paradox model that

we discussed previously. The random inputs are 8 characters long with 62 possible characters for each of the 8. Thus, the number of possibilities is 62^8. For the number of inputs generated, we will use 300,000 because that is 4 times our expected average. $1-\prod_{x=0}^{299999}\left(\frac{62^8-x}{62^8}\right)\approx 0.000206$, so the probability of generating the same input is minuscule.

Findings & Analysis

I executed my program 20480 ($2^{11}\times 10$) times in order to find an experimental probability that was semi-longterm. The program recorded each trial in an excel spreadsheet for easy storage and later analysis. I will not include all of my data due to sheer length; however, here is a sample of my data (first 5 trials):

Trial #	String 1	String 2	Hash	# of Hashes Generated
1	HC7sivAN	I45m80zo	23E07A67	87250
2	4WDBTwog	D0H0qUvS	104F7337	87027
3	diUtquDh	vus5SBmE	2979A97C	106844
4	M7yqoLCv	rCIAKblt	2A165DC1	74139
5	32k7EqOJ	3khgMfdi	38BFF5AF	183902

The first things I did after all of my data was collected was to calculate some summary statistics on the *# of Hashes Generated* column because I wanted to find the number necessary to find a collision.

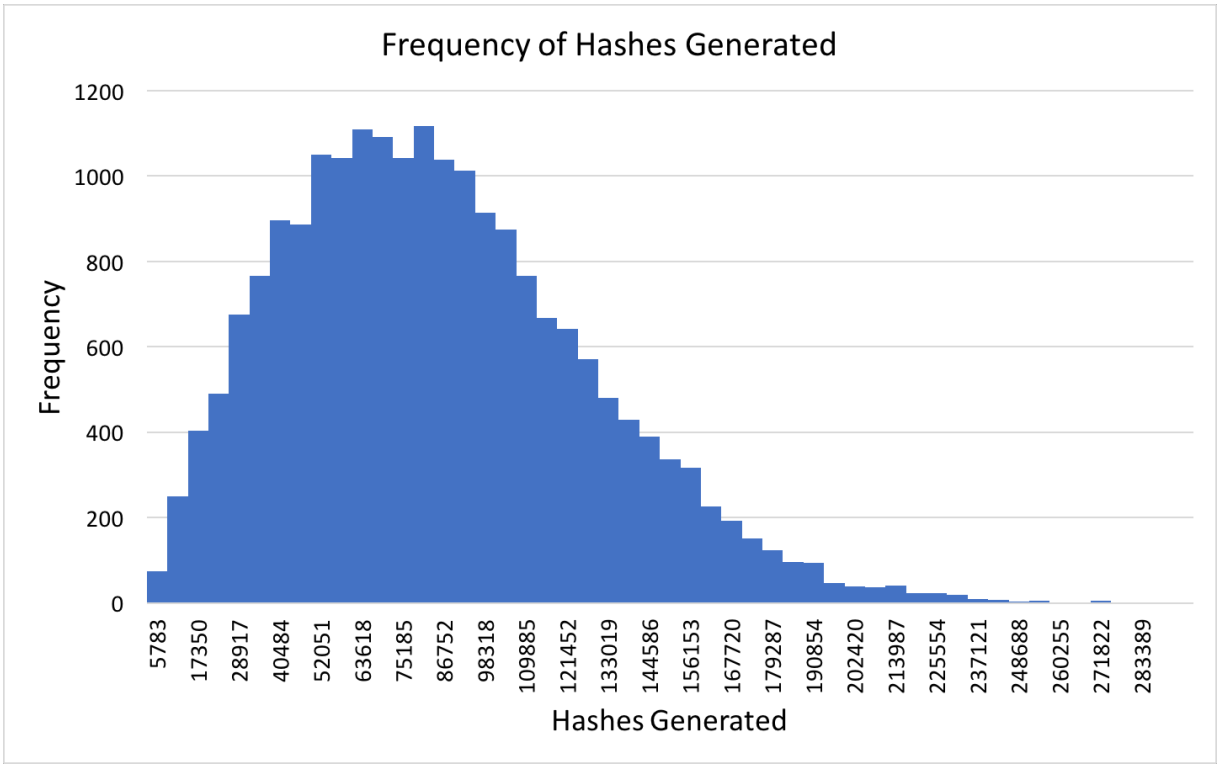
Here are the summary statistics that I found:

Mean	82110
Median	77613
Range	288557
Minimum	615
Maximum	289172

My initial thought was that the mean of 82,110 was not that close to my expected value using the approximated model ($n = 77{,}163$). However, the median was much closer to that value at just 450 off. Upon closer examination, it makes more sense that the median is closer to our expected value. If we used 50% as the probability in our initial prediction, then approximately half of the trials should be below the expected value, and half of the trials should be above it. This happens to be the definition of a median. We can take our experimental median and plug it into our original model to get our experimental probability of finding a collision. Thus, $1-\prod_{x=0}^{77612}\left(\frac{2^{32}-x}{2^{32}}\right)=0.504029$, or 50.4%. This percentage means that in the experiment, it took slightly more inputs to find a collision than predicted by the model. If we take the difference between the predicted and the actual number, and we then divide by the predicted number, we get a margin of error of about 0.6% ($\frac{450}{77163}=0.00583$). Overall, however, the experiment checked out. The actual number of hashes taken to find a collision was indeed approximately the number that we predicted using the mathematics.

Once I saw that the median was close to the value that I predicted, I went back to the mean. I wondered why the mean was noticeably different compared to the median. I figured that it had something to do with the distribution of the trials based on my basic statistics knowledge.

To confirm this hunch, I created a histogram in excel to graph the data:



My hunch was confirmed. The graph showed that the data has an obvious right skew. Therefore, it is not appropriate to use the mean because the data is not normally distributed. Skewed data pulls the mean away from the median. (I recognize that the horizontal scale appears quite awkward; however, I derived the seemingly random x-axis labels by dividing the maximum x-value—289172—into 25 bins.)

Conclusion

With the help of a computer program, I was able to see for myself that the birthday paradox can indeed be applied to cryptographic hashing. The predicted number of hashes needed to produce a 50% chance of a collision was very similar to the number predicted with the mathematical model with only a 0.6% difference.

There is a specific reason that we used 50% in finding the number of hashes needed to produce a collision. This probability is important because it is the point at which the generation of hashes begins to be unproductive. If there is more than a 50% chance of generating a collision, then a collision will occur more times than not, and producing a collision defeats the purpose of hashing the data in the first place—to keep the information private. In the words of an expert, the hashes are no longer“useful” after the 50% mark is reached because a collision is imminent and the input data is less relevant (Azad).

It is important to note that the CRC32 algorithm used in this exploration would not actually be used in production to safeguard important information due to the limited length of the hash. According to an article by a security software company, all modern cryptographic hashing algorithms have at least a hash of 128 bits or more (Persits Software, Inc.). CRC32 was used to easily demonstrate the simplicity of finding a collision, an issue that current algorithms with much larger hashes still potentially face. Using the mathematics of the birthday paradox, we now understand how easily the integrity of private data could be compromised due to the potential for collisions to occur.

Works Cited

Azad, Kalid. "Understanding the Birthday Paradox." *BetterExplained*. N.p., n.d. Web. 18 Sept. 2018. <https://betterexplained.com/articles/understanding-the-birthday-paradox/>.

"Birthday Problem." *Wikipedia*. Wikimedia Foundation, 12 Sept. 2018. Web. 18 Sept. 2018. <https://en.wikipedia.org/wiki/Birthday_problem>.

Hashem, Rayiner. "CRC32 Hash Collision Probability." *Ars Technica*. N.p., 22 Jan. 2008. Web. 18 Sept. 2018. <https://arstechnica.com/civis/viewtopic.php?f=20&t=149670>.

Persits Software, Inc. "Crypto 101: One-way Hash Function." *AspEncrypt.com - Crypto 101: Basic Terminology*. N.p., n.d. Web. 18 Sept. 2018. <http://www.aspencrypt.com/crypto101_hash.html>.

Pierce, Rod. "Taylor Series." *MathIsFun.com*. N.p., n.d. Web. 18 Sept. 2018. <https://www.mathsisfun.com/algebra/taylor-series.html>.

Preshing, Jeff. "Hash Collision Probabilities." *Preshing on Programming*. N.p., 4 May 2011. Web. 18 Sept. 2018. <http://preshing.com/20110504/hash-collision-probabilities/>.

Appendix A: CollisionFinder.java

```
/*
 * Copyright (c) 2018 <Candidate gzk379>. All rights reserved.
 */

import java.util.ArrayList;
import java.util.Collections;
import java.util.zip.CRC32;

public class CollisionFinder {

    public static void main(String[] args) {
        findCollisions();
    }

    //lists of attempted inputs (strings) and their outputs
(hashes)
    private static ArrayList<String> strings = new ArrayList<>();
    private static ArrayList<Long> hashes = new ArrayList<>();

    //begin meaningful code execution
    static void findCollisions() {

        //creates object that contains algorithm, takes input to produce hashes
        CRC32 hasher = new CRC32();

        //add one default value (0) to array of outputs
        hashes.add(hasher.getValue());

        //loop until collision found, max 2^32-1 (same amount of possible hashes)
        for (int i = 0; i < Integer.MAX_VALUE; i++) {

            //generates random string (random sequence of characters) to input to
hashing algorithm (length of 8 characters)
            String currentString = randomStringGenerator();

            //inputs random string into CRC32 algorithm
            hasher.update(currentString.getBytes());

            //gets output of CRC32 algorithm (the hash)
            long hash = hasher.getValue();

            //resets algorithm for next hash
            hasher.reset();
```

```
             //checks if collision found
            //if no collision, code attempts again starting at line 27
            if (!insertInOrder(hash, currentString)) {
                //executes when collision found

                //finds first string that produces same hash as current string
                //only executes after collision found (that means there is always two
inputs that produce the given output, this code finds the first one)
                int otherIndex = Collections.binarySearch(hashes, hash);
                String firstString = strings.get(otherIndex);

                //confirms outputs match for both inputs
                hasher.reset();
                hasher.update(firstString.getBytes());
                long confirmHash = hasher.getValue();

                //displays information about the collision to record in a spreadsheet
               System.out.println("Collided hash:\t" + currentString + " --> " + hash
+ "\n\t\t\t\t" + firstString + " --> " + confirmHash);
                System.out.println("Number of attempts:\t" + (i + 1) + ", (First
occurrence: " + (otherIndex + 1) + ")");

                //exit program
                break;
            }
        }
    }

    //possible characters used when generating random strings
    //these characters used to avoid the occurrence of the same string being generated
twice
    //  62 characters means 62^8 = 218,340,105,584,896 possible combinations
    //  nearly 0 chance (hundreds of decimals places of 0s) of generating the same
string twice using the same birthday paradox formula
    private static final String ALPHA_NUMERIC_STRING =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    //returns a string of random characters of a specified length
    //from https://dzone.com/articles/generate-random-alpha-numeric
    private static String randomStringGenerator() {
        int size = 8;
        StringBuilder builder = new StringBuilder();
        while (size-- != 0) {
            int character = (int) (Math.random() * ALPHA_NUMERIC_STRING.length());
            builder.append(ALPHA_NUMERIC_STRING.charAt(character));
        }
        return builder.toString();

    }
```

```
    //checks if collision has occurred (i.e. if output has been produced before)
    //otherwise, inserts input and output into their respective lists
    private static boolean insertInOrder(long element, String string) {
        for (int i = 0; i < hashes.size(); i++) {
            if (element == hashes.get(i)) {
                //collision found
                return false;
            }
            if (element < hashes.get(i)) {
                hashes.add(i, element);
                strings.add(i, string);
                return true;
            }
        }
        hashes.add(element);
        strings.add(string);
        return true;
    }
}
```