**Programming Project 2 – The Birthday Attack**

Karxyriah Ashley and Chance Currie

The Task

The task is to run a program that runs the “birthday attack” against a hash function that is defined as “BadHash44.” This hash is a short hash function that is constructed through SHA256. BadHash44 will only take the first 44 bits of the output that is given by SHA256, and our main objective is to find two arbitrary inputs of the function BadHash44 which result in the same output. In other words, we need to find a “collision”, which is the same hash being used for two different inputs, in the BadHash44 function. We also need to write the random message inputs and corresponding hash values into a text / .csv file and it needs to be sorted based on hash values.

The Program

Our code was written in Python. The program begins with importing a couple of important Python libraries: hashlib and random. For hashlib, we import this to utilize a built-in implementation of SHA256. For random, this library will be used for generating random messages that will have their hash values compared for collisions.

Next is the previously mentioned BadHash44 function. It takes in a plaintext message, encodes the message, then creates a SHA256 hash of the encoded message through the implementation of the hashlib library, and the function will return only the first 44 bits of the output that is generated from the SHA256 hash.

We then have the find\_collision function, which is where the meat of our program is. First, we define variable n to be the length of a SHA256 hash, which, given by its name, is 256 bits, so we set n to be 256. We then open a file called hash.csv (we already have this file created on our machine, but no information is currently contained within it) with the intention to write to the specified file. We then define variable x0 to be a randomly generated number in bits with the size of n, where n is 256. Then, we convert x0 from bits into a hexadecimal value, and we want to get everything from index position 2 to the end of the newly converted hexadecimal string value. While the length of x0, being the hexadecimal string in this case, is not equal to 64, we want to add the number 0 to the front of x0 in string form. When then set variable “x” to be the value of x0, and then set variable “x\_prime” to be the value of x.

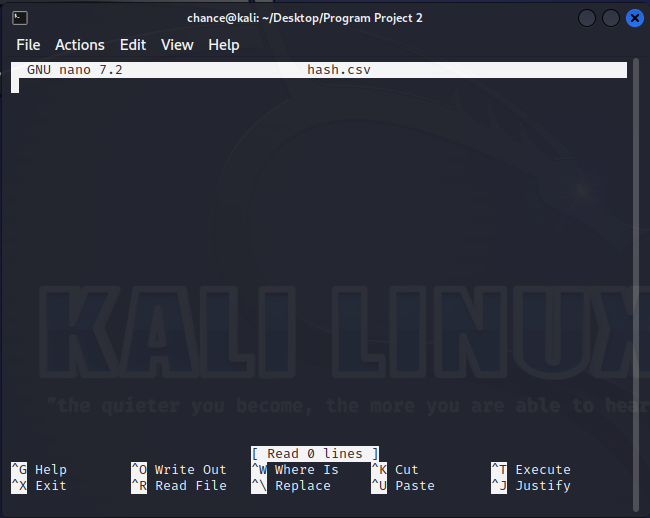
The variable “i” will be our variable that will update as we go through our while loop, and it’s initialized with a value of 0 to begin with. Our while loop that follows the initialization of i is as follows: while the variable i is less than 2 to the power of n/2, we run the BadHash44 function with the message being x. Once the BadHash44 function has finished running, we set its return value to be the current value of x. With the x\_prime variable, it’s the same idea, but the BadHash44 function is ran twice, so it would look like: BadHash44(BadHash44(x\_prime)), and this is set to the current value of x\_prime. We then need to compare and see if x is equal to x\_prime. If they are the same, then we break the while loop, otherwise we increment the value of i by 1 and run the while loop again.

For each iteration of the while loop that is ran, we want to write information to the file that we opened earlier. To start, we set x\_prime equal to x, and set x equal to x0. Then, we have a for loop that will run for j in the range of i. Each pass through the loop, we want to write the value of x and its corresponding BadHash44 has value, as well as x\_prime and its corresponding BadHash44 hash value. If the BadHash44 hash values of x and x\_prime are equal, we want to print out those hashes respectively and close the file. We then want to return x and x\_prime. If they are not the same, we want to make x equal the BadHash44 hash value of x, and make x\_prime equal to the BadHash44 hash value of x\_prime. If we find that none of the BadHash44 hash values equal one another, we return None, indicating that no matches were found.

At the bottom, we want to test the function, and we do this by calling the find\_collision function on x and x\_prime. If x returns None, the we print out a statement saying that no collision was found. However, if there is a returned collision, we print out a statement saying that a collision was found, and we print those values respectively.

Program Demonstration Through Screenshots

Using the nano editor inside of Kali Linux, we create a file called hash.csv, and currently, there is nothing inside of it. This file is also located within the same directory as the main.py program.



Here, we are running the program. There won’t be any output for a little bit, probably no more than a minute depending on your hardware, as the program is working behind the scenes to find any collisions.

Text

Description automatically generated

Here, we have found that there is indeed a collision. We can see that the hashes of x and x\_prime are the same, despite the numbers of x and x\_prime bring the same.

Graphical user interface, text

Description automatically generated

Upon completion of the program, if we now open the hash.csv file, we can now see that the file has been populated with data. The screenshot only shows the beginning portion of the data.

Graphical user interface

Description automatically generated