Hash Function Description
My hash function works in several stages. I will describe each of these stages in detail below.

Initialization
In this stage, the variables xor_tot and sum_tot are initialized to 32 bit random values. The 64 bit variable large_dig is also initialized, and will hold the larger version of the digest from which a section is selected for outputting

Combination
In this stage, the xor_tot and sum_tot are calculated. xor_tot is calculated by xoring the current value with the next 32 bit chunk until no chunks remain. Sum_tot is calculated by adding the current value to the next 32 bit chunk until no chunks remain.

Large Digest Generation
The large digest is calculated in parallel with the combination stage. After finding the new value of xor_tot and sum_tot for each chunk, each of these values is xored and added to every position of large_dig. This means that for every chunk, we perform large_dig ^= xor_tot << (0 to 31), large_dig += sum_tot << (0 to 31), large_dig += xor_tot << (0 to 31) and large_dig ^= sum_tot << (0 to 31). At the end of every chunk, large_dig appears relatively random.

Digest Selection
To select a chunk of large_dig to use in the output, large_dig is right shifted by large_dig % 32 bits, and the least significant 32 bits are selected as the digest.

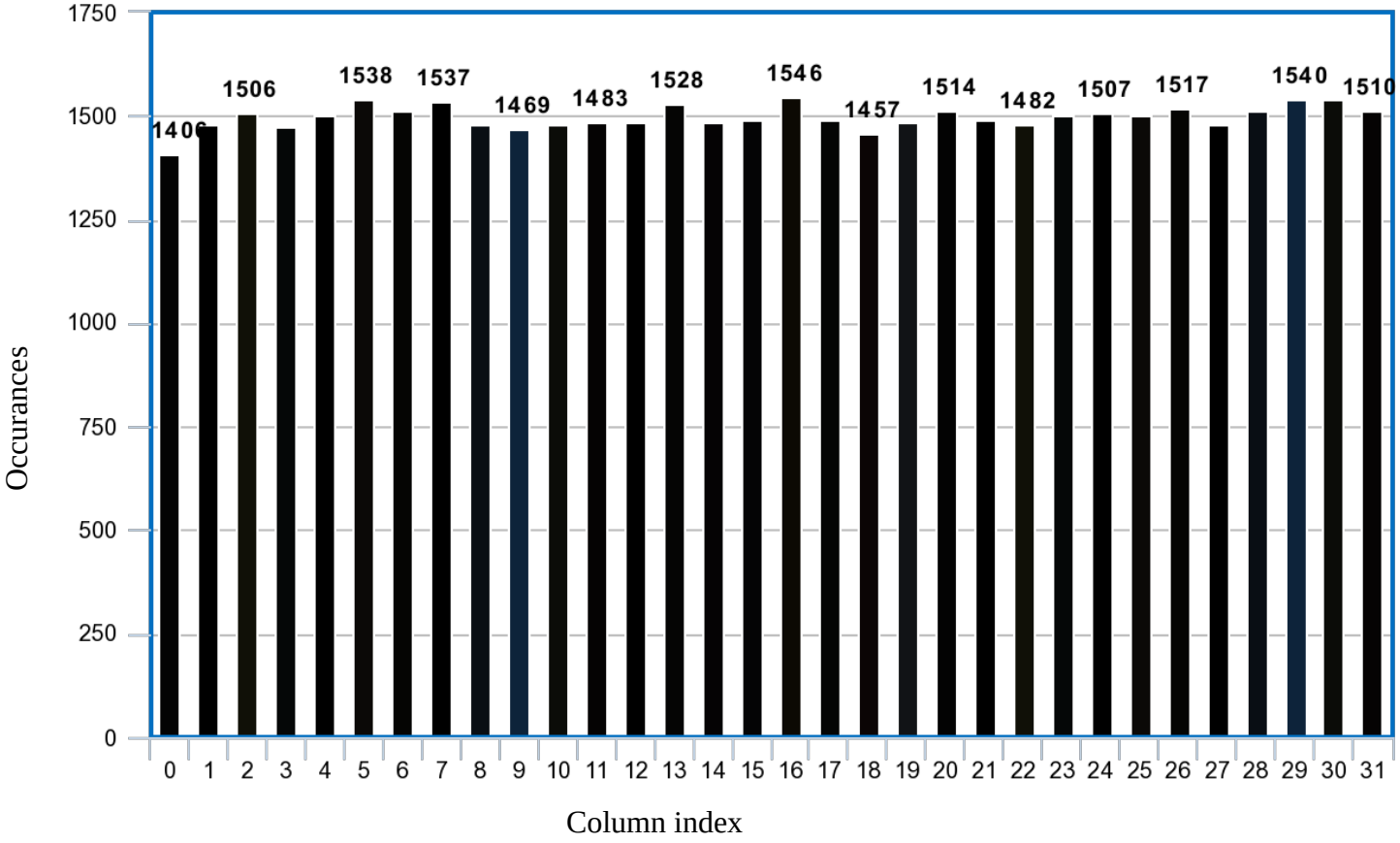
Shuffling
One of four possible shuffling options is selected by taking large_dig % 4. Each byte of the digest is placed randomly based on the selected shuffling option.

Output
The final output of the hash function is the xor output of the digest, the final xor_tot and the final sum_tot.

Unique Digest Values
1. There are no collisions in the original files (Other than the zeros, but this is because they are all the same file)
2. There are no collisions between any of the hashes in the mod directories.
3. Since no collisions occur, there are no issues to discuss in this case.

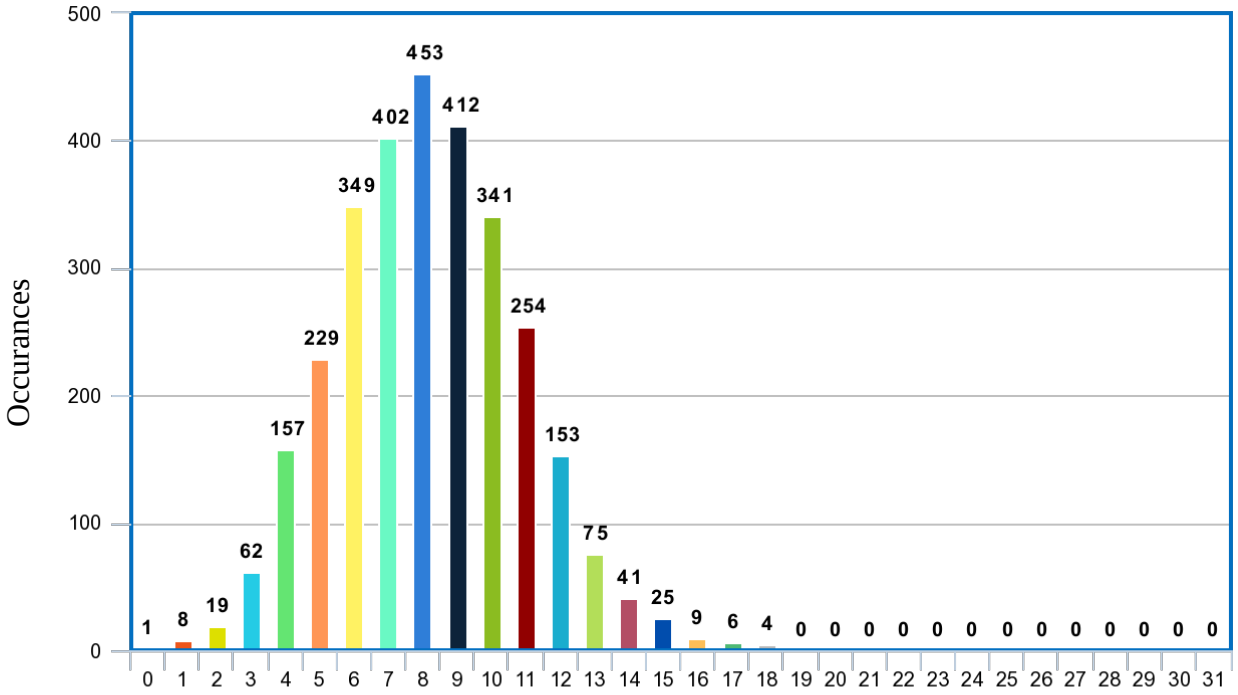
Ratio of 1s to 0s
1. shak – avg: 0.5032
zero – avg: 0.5002
random – avg: 0.4945
2. Standard Deviation of all: 0.004
3. These results say relatively good things about the algorithm. The results are all within about 0.5% of expected with a low standard deviation, meaning the algorithm produces nearly even 1s and zeros.
4. Average: 0.4945
5. Nothing specific was done to combat imbalances, but the idea for the large digest was inspired by the LFSR algorithm, which helped me to produce random looking hashes.

Ratio of 1s to 0s in columns



1. These results again reflect well on the algorithm. In a perfect hash, every column would be right around 1500. The main concern I have is column 0 which is pretty low compared to most other columns.
2. Once again, I would say the LFSR inspired large hash helped to spread the values evenly through the columns. It's possible the sum_tot and xor_tot tend to have certain values in column 0, causing that column to have a lower than expected ratio of 1s to 0s.

Average Difference Between Mod and Original



# of matches between mod and orig

1. I'm only including the graph for all of the mod/orig comparisons together, but the same bias exists in all the files. This strong skew to the left strongly implies that my hash function is vulnerable to preimaging, meaning that only changing one bit changes significantly more than half the bits. If we were to invert the mod hashes, we would have significantly more collisions than we would want.
2. There are no outliers, except for the fact that the entire distribution skews towards fewer matches. Other than that, it's a very regular looking distribution.
3. This is not a good output, it shows that my algorithm has issues preventing preimages.

Reflection

1. Is the hash collision resistant?

No, while I experienced no collisions in the test data, smhasher found many collisions. To me, this implied that there are large sections of the 32 bit space which my hash function is not utilizing.

2. Is it preimage resistant?

No, the skew we can see towards fewer matches when comparing orig to mod shows that we are not following the strong avalanche criteria. This means that if you wanted to find two inputs which hash to the same output, changing as few bits as possible, then inverting the output would get you too close for comfort. One output even had zero matches, which means it was just the inverted orig hash.

3. If you had more time, how would you improve your hash?

I would add more shuffling steps, potentially doing it programatically to allow every permutation of the 32 bits to be possible. I would also remove the final xors with the sum_tot and xor_tot as I believe these may play a role in the preimage issues I'm experiencing.

4. Was the project interesting?

Very, I realized how difficult it would be to actually write a good hash function, and I gained the ability to evaluate one myself.

5. How would you improve the project?

Running make with -O3 may have significantly sped up my smhasher run times, but I did not verify that.

Additional Information
Smhasher results can be found in Results.txt
I have added a README.txt file to the zip file which explains how to run it.
The scripts I used to do my analysis can be found in the analysis folder, along with data modified to remove the filenames.