Tony Alexander

Advisor: Randy Appleton

Committee: John Sarkel & Hadi Shafei

Analyzing the Randomness of the Keccak 512 Hashing Algorithm using a Provably Fair Gambling Site

My project is about analyzing the Keccak 512 hashing algorithm by plugging in a random hash, running it through the gambling site WTFskins.com equation, and determining the cash/risk ratio of running consistent bets over the span of roughly a year. So what does this mean exactly? Let's take a few steps back to determine how I got to this point.

Background

One of my hobbies is playing video games. Counter Strike Global Offensive is one of my favorite and because Steam (Video Game Distribution Service Client) allows your in-game inventories to be tradable to other players with cash or other in-game items, it has in turn created many sites that deal with the buying, selling, trading, and gambling of these items as a business. One of these sites is WTFSkins.com. Upon clicking this link you will see a game called Crash. There is a multiplier that starts at 0.00x and exponentially increases as time goes on. Your goal is to bet on a multiplier that you think it will hit before the inevitable crash happens. For example, you bet one dollar on the multiplier hitting a 10.00x. This has roughly a 1 in 10 chance of happening. If it were to crash at 4.67x, then you would lose your dollar. Otherwise if the multiplier crashed at or above a 10.00x, then you will win ten times your total bet. On the bottom of the screen, you can see a hash value that changes upon every round of crash. This hash value is the provably fair data. To calculate this hash value, they create a hash based on the day of the month, the round ID, and the hash that will come 1 round after this current hash. So in theory, if you knew the first hash value that was created, you could predict every crash number and become the first trillionaire. However, your chances of cracking it are basically zero.

First Test

Upon seeing all this information, I decided to grab from their history page the last 1 million crash multipliers to start analyzing data. I noticed that if I went back as far as possible on the history pages, then 1 million crash numbers were associated with roughly a year of data. My goal was to look for patterns and repetitions to see if one bet strategy might be better than another. I wrote a script in python that web scrapes all these numbers and puts them into a text file.

For analyzing these numbers, I decided to go for an approach to use a martingale betting strategy. There were also three variables that I took into account when testing different combinations of bets. The first variable that I used was the multiplier that I was looking to achieve. The second variable was the scale at which to increase my bet by. And finally, the third variable was how big of a gap between wins when I would increase my bet by a specific scale. So, let's say I have a multiplier of 10, a scale of 5, and a gap of 3. The graph below shows how this combination of numbers would work.

Round	Bet (Starting $1)	Total Risk	Win/Lose

1	1	1	Lose
2	1	2	Lose
3 (Gap reached)	1	3	Lose
4	5	8	Lose
5	5	13	Lose
6 (Gap reached)	5	18	Lose
7	25 (Reset bet to 1)	43	Win $250, Profit $207
8	1	1	Lose
9 onwards	Repeat as above	Repeat as above	Repeat as above

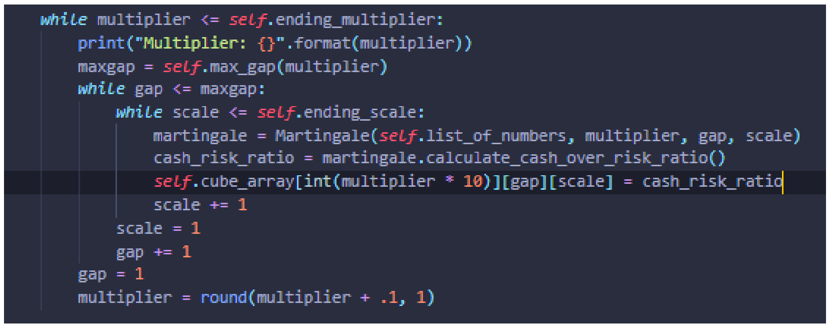
If I iterate over all of the crash numbers that I grabbed from their website and tried a whole bunch of different combinations of multiplier, gap, and scale, then I could calculate a cash/risk ratio that shows the most I ever risked on the 1 year of betting compared to how much money I made. The goal would be to find the highest cash over risk ratio, since you would be risking the least to make the most amount of money.

Because one of my main goals was to visualize a representation of cash/risk ratio values, I knew I could make a 3d array where the x, y, and z axis would correlate to the multiplier, gap, and scale. The value at each of these coordinates would be the actual cash/risk ratio. With a 3d array, we can now visualize the data on a 3d plot and have the cash/risk ratio be correlated with a specific color set on the plot. This is my current end goal. Visualize all of these different values to see if there is a positive money making scheme that is based on the hashing algorithm.

Improvements

After completing a rough draft of all these steps mentioned above, I noticed that there were plenty of combinations of a multiplier, gap, and scale that were giving positive cash/risk ratios over 1 year of consistent betting. This really shouldn't be the case however, since in gambling, *the house always wins.* Finding this information out was exciting but also concerning, since either there was a possibility to make a lot of money or something was wrong. This led me to a couple of theories. The first is that the hashing algorithm is not sufficient in creating a pseudorandom hash to be fed into the website formula for calculating a crash point. This is highly improbable however since there is no other proof that the Keccak 512 hashing algorithm is not a good hash from other research, so this should not be the case. My second theory was that the website's formula that you feed the hash may have a bug in it, but this also can be disproved since they give the source code for this on one of the pages (I also was unaware that they provided their formula until further digging on their website). The third theory was that I miscalculated how I was determining the cash/risk ratios. I started looking further and noticed that my code was fine. I was stumped as to what could be wrong. To try and solve the issue I decided the best course of action would be to start generating my own crash numbers based on the website's formula and running my cash/risk ratio script on generated data instead of what the website gave me. Since the first test was with 1 million numbers, I decided to scale it up to 30 different text files each with 1 million crash multipliers in each. This gives me a much more averaged out data set to work with and can also get rid of many inconsistencies. I ran the multipliers, gaps, and scales that were returning high cash/risk ratios from the first data set on my newly generated data set files and the results were very inconsistent. Some combinations of the three variables would vary greatly between each file which proved that my data set needed to be much larger than just looking at the history that the website gave me of crash numbers. Despite solving why so many numbers were returning positive results when analyzing the website's crash history, it was disappointing since I knew the odds of making money were not nearly as good. The other problem that I ran into was having such a large data set was causing the code to take much longer to run through my script.

Running a data set file on a single multiplier, scale, and gap at this point takes half a second to test one single data file. This is not nearly as efficient as I would want it to be. The code needs to be optimized and using Python does not help. I would have switched to a different language but Python allows me to handle complicated parts of code easily (webscraping, file IO, plotting, etc.), but it's just not fast. Another reason that I want to increase the speed is because I planned on adding a script that will start at a base multiplier, scale, and gap and increment 1 of these variables up until a maxpoint. An example is shown below.



```
while multiplier <= self.ending_multiplier:
    print("Multiplier: {}".format(multiplier))
    maxgap = self.max_gap(multiplier)
    while gap <= maxgap:
        while scale <= self.ending_scale:
            martingale = Martingale(self.list_of_numbers, multiplier, gap, scale)
            cash_risk_ratio = martingale.calculate_cash_over_risk_ratio()
            self.cube_array[int(multiplier * 10)][gap][scale] = cash_risk_ratio
            scale += 1
        scale = 1
        gap += 1
    gap = 1
    multiplier = round(multiplier + .1, 1)
```

Because the Martingale class takes .6 seconds to run (since it has to increment over 1 million digits to calculate cash/risk ratio), and I plan on running this thousands of times on different multipliers, gaps, and scale across 30 files, I have estimated out that a multiplier from 5-10 (increment .1) with a gap of 0-maxgap and a scale of 1-25 will run Martingale roughly 125,000 times for a single file. The time estimate on this is roughly 18 hours for one data file or 22 days for all 30 data files. This is why my goal was to utilize GPU cores using Cuda to speed up this time. However, after further research I came across a Python library called Numba that will take a method or class and convert it into optimized machine code at runtime. This saved my life. I added support for the Martingale class to use Numba and instead of taking 18 hours for one data file, it now takes 22 minutes. This is a speed up of roughly 50x or an increase of 5000%. This lets me generate 3d arrays for all 30 files in roughly 12 hours. That is equivalent to analyzing well over 3 trillion multipliers!

Plotting

Now that I finally tested all 30 data files and created a 3d array for each of these files, I was able to plot the data. I used the Matplotlib Python library for this since it allows easy visualization of data. There were two different features that I wanted to be able to graph.

The first feature is graphing a single data file onto a plot. This means I can have 30 different plots all with different data. I would be able to compare and contrast each of the plots to see if there are recurring patterns within each. If so, then we know that there is a consistency. Upon creating the first plot, I realized how slow it was to move and rotate the image. This was because I was graphing 1000's of points and my cpu was having a hard time keeping up. I realized that there was no need to plot any cash/risk ratios that are negative, since in those cases, you would be losing money. To add onto this the plot was also way overpopulated with points. In order to fix this, I ignored all cash/risk ratios that were 0 or less and instead only populated the plot with positive ratios. This was much better for my computer to handle, and also showed a recurring pattern among all data files where you can make money!

The second plotting feature I wanted to add was being able to create 1 plot based on averaged ratios from all 30 data files that I created. This in turn would show me how much I could expect to earn over the course of 1 year using an average of 30 years for reference. To my surprise, some combinations of the three variables proved to return a 1.5 cash/risk ratio on a consistent betting strategy over 1 year!

Automatic Betting on WTFSkins.com

After doing all of this big data analyzing, my final step was to create a script that will automatically place bets for me on the website. This will include navigating to the webpage, placing a bet and changing bet size at specific points set by scale and gap. This was done using the Selenium Python library to web scrape html elements and click buttons, change values, etc. I also had to utilize a Chrome-based webdriver to allow this automatic script on Chrome.

Concluding Thoughts

This project has been roughly two years in the making, and being able to visualize my discoveries is exciting to me. Because my main goal was to analyze the Keccak 512 hashing algorithm indirectly through this website's formula, I suppose I have to have an answer to my findings. From what I have found, the vast majority of cash/risk ratios do tend to be negative values where you are losing money. However, there are consistent patterns that do occur where you have a positive cash return over very long periods of betting. So, the algorithm mostly remains consistent in being random, but as to how random of values it returns, I will let the visualizations show to you how random or not-random you think it is (I will be showing the visualizations in my presentation). In my opinion, I believe that the hashes it returns do have some credit, but computer randomness is also something that is hard to prove. Finally, if I had to wrap up what I learned through this whole process, it would include, Python optimization, working with big data, web scraping, visualizing important aspects of big data, and creating graphics.