The Creation and Optimization of a

New Cryptographic Hash Function

By Monica Ingham

Table of Contents

Abstract…………...……………………………………………………………....2

Introduction……...…………………………………………………………….….3

Cryptographic Hash Function Analysis………………………………...….…..4

Research Methods……………………………………………………………….6

Algorithm Framework………………………………………...………………….9

Part 1: RNG Matrix Initialization…………………………………...………….9

Part 2: RNG Matrix Value Swapping…………………………………………12

Part 3: Message Digest Creation…………………………………………….13

Algorithm Testing and Optimization…………………………....……………..15

Part 1: Algorithm Speed Test…………………………………....………..…16

Part 2: Message Digest Character Bias Test………………………....……...17

Part 3: Avalanche Effect Test………………………………………………..18

Conclusion……………………………………………………....……....…..…..19

Works Cited…………………………………………………………………..….21

Abstract

As the power of computers increases over time, it creates an ever present demand for the creation of new cryptographic hash functions (CHFs) that are more secure. For this project, I designed a CHF based on the stream cipher RC4, a simple but robust algorithm designed for secret key encryption in 1987. This new cryptographic hash function takes the input of a string of characters with any length, and performs a one-way encryption that creates a 128 bit digest. It implements the same basic three step process that is also used in RC4: initialization of an array, then swapping all of the values in the array at random, and finally using this array to create a pseudorandom number generator.

The cryptographic hash function is primarily based around the concept of an RNG Matrix, a design I created for a 32x32 two dimensional array of pseudorandom numbers. Using this concept combined with the three step process from RC4, it first expands the input into data in the RNG Matrix, then swaps every value position at random, and finally condenses all 1028 numbers into a single 128 bit output to create a satisfyingly random message digest. The CHF performed very favorably in the tests used to gauge how secure it is, with the algorithm obtaining near perfect scores in the two most important tests. However, there is still much more testing that is required to be completed before the cryptographic hash function can be deemed secure, which is something that could not be achieved in the very limited time frame for this project. But regardless of how well the algorithm might fare in these tests, the concept for the RNG Matrix is very versatile and could be utilized to create a much more capable CHF.

Introduction

Before starting, it’s necessary to explain what a cryptographic hash function is, and why research into it is relevant to progress the field of cryptology. Wikipedia defines a cryptographic hash function as “…a mathematical algorithm that maps data of arbitrary size…to a bit array of a fixed size…It is a one-way function, that is, a function which is practically infeasible to invert or reverse the computation”. What this basically means is a CHF takes any string as an input, such as a username or password, and then outputs a digest of fixed size that is ideally impossible to reverse. For example, the cryptographic hash function SHA-1, given the input “The red fox jumps over the blue dog”, creates the digest “0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC”. It is simple but versatile type of algorithm, with many different uses such as: password verification, electronic signature generation, file integrity verification, and many more. In fact, it is so versatile that in its field, it is regularly referred to as “the swiss knife” of the cryptology toolkit (Kishore and Raina, 504).

Despite cryptographic hash functions being developed and optimized for over thirty years, there will always be a need for improvement. As computers become more powerful over time, current CHFs become easier to crack year by year just by brute force attempts. Of course, while people are making new algorithms, there are also people constantly discovering vulnerabilities in existing algorithms that create a demand for the new hash functions. As Kishore and Raina put it, “Hashing continues to remain a slow and computationally intensive process, as the gains due to the expansion in computing power have been offset by the ever increasing magnitude of data that need to be hashed. Moreover, CHFs have become increasingly susceptible to sophisticated attacks, with attackers employing high-performance computing (HPC) for cryptanalysis” (505). Eventually, every CHF that exists currently will be cracked in due time, making the demand for new and improved cryptographic hash functions ever present. The first notable hash function, MD5, was created in 1991 and was considered very secure until 2005, when it was declared broken and unsafe for commercial use (Whittaker). With the computers we use today, a simple brute force collision attack can be done in seconds. With current algorithms constantly becoming obsolete over time, research into cryptographic hash functions will always be warranted.

Cryptographic Hash Function Analysis

Another important topic that must be covered before continuing is a simple question: what properties are a cryptographic hash function comprised of that it must fulfill? The first property is that a CHF must be deterministic; that is, an input must always generate the same output, no matter what. If it creates a random output, then it is simply a random number generator that uses a seed as an input, not a cryptographic hash function. The second is that a CHF must be a very fast algorithm, since it must be able to be run on any device, including very old ones. Thirdly, it must be resistant to pre-image attacks, meaning if you are given an output, it has to be very difficult to find an input that creates that specific output. Finally, it must be resistant to second pre-image attacks, also known as weak collision; this means that given an input, it must be difficult to find a second input that creates the same digest as the first.

These are very good and definitely important properties of cryptographic hash functions; however, apart from algorithm speed, they are not what people usually focus on when researching CHFs. One of the most important aspects of a cryptographic hash function is that it is required to be resistant to collision, which means that it must be very difficult to find two different inputs that create the same output. This is very similar to second pre-image resistance, but it is possible to have a CHF that is collision resistant while being susceptible to second pre-image attacks. The last property of CHFs is the avalanche effect: this states that even if you change a single bit of the input, the message digest must radically change (see fig. 1). If the avalanche effect does not exist in the algorithm, then with some analysis, a person could theoretically make predictions for what the digest might be for any given input, therefore rending the CHF insecure. Every cryptographic hash function must have all six of these properties; if even a single one is insufficient, then the algorithm is considered obsolete and should not be used.

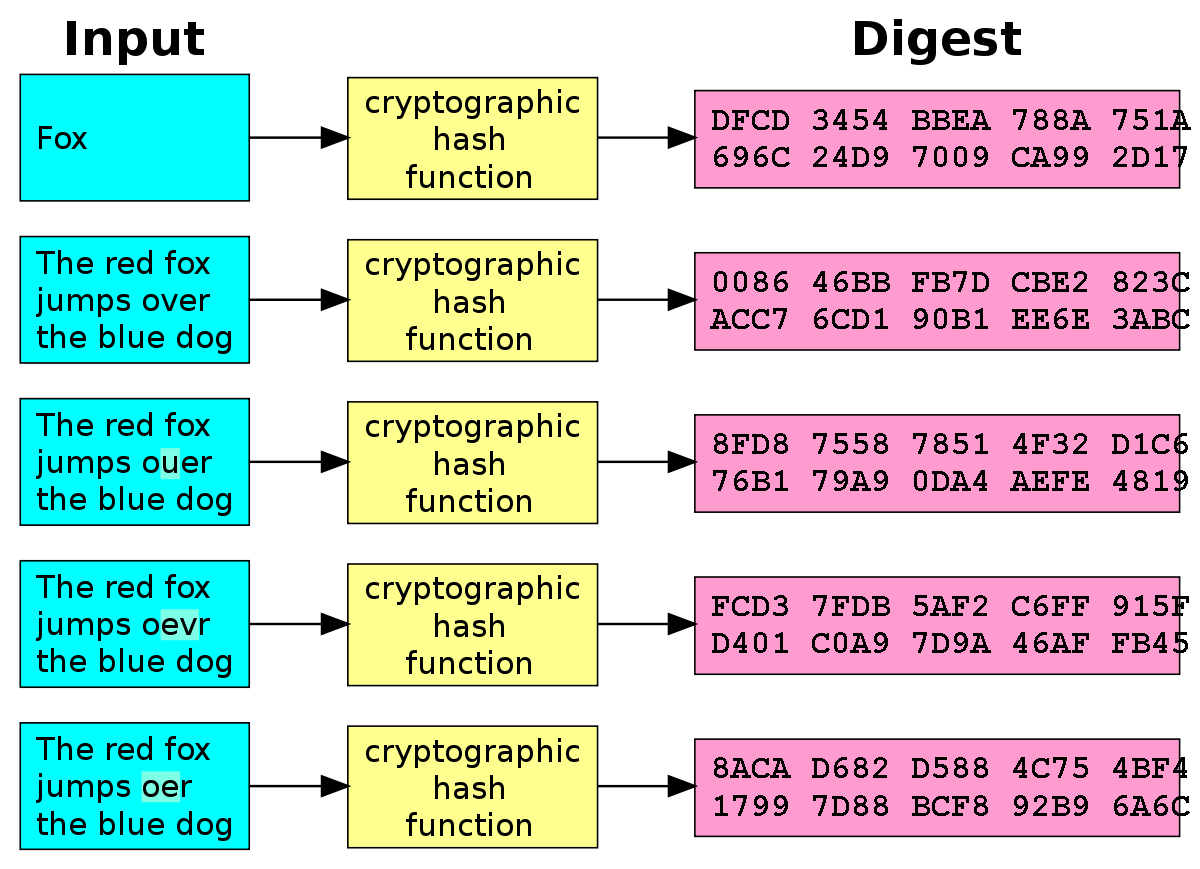


Fig. 1. A visualization of the avalanche effect, and how changing a single bit of the input

should drastically change the message digest. Image source: [wikipedia.org](https://en.wikipedia.org/wiki/Cryptographic_hash_function)

Research Methods

Since most of this project was an idea of mine for a cryptographic hash function based on the stream cipher RC4, almost all of the research was for understanding this algorithm and how I can use similar methods from RC4 to improve my own algorithm. With the very small time frame for this project taken into account, there was almost no time for any research on other subjects; therefore, this section will only focus on my research into RC4. The stream cipher is quite old, having been released in 1994, however, “…after going through significant analysis by researchers, [RC4] proves to be robust enough on different platforms” (Das, et al. 1). For this reason, there have been many cryptographic hash functions based on this algorithm, as its simplicity serves as a great starting point.

RC4 is a very simple algorithm that is comprised of three main steps. First, it initializes the array S, which has 256 elements initialized with the values from 0 to 255, in incremental order. Second, it uses information from the key (RC4’s intended use is a secret key encryption algorithm) to pick random locations and swaps each element of the array S with another, thereby jumbling the order of all the values. Once this is finished, it moves on to the final step, where it uses S to create a stream of pseudorandom numbers. For the cryptographic hash function I created, I decided to start by adopting this three step process, and see what I could create using this tried and true method.

For the first major difference between RC4 and the CHF I created, I used the idea behind the array S and created something I call the Random Number Generator Matrix, or RNG Matrix. Where RC4’s S is an array of the first 256 numbers, the RNG Matrix is a 32x32 two dimensional array of pseudorandom numbers. The reason I chose to make the RNG Matrix two dimensional is that for the third step, which condenses all the numbers into a 32 digit hexadecimal digest, it creates many different possible strategies and algorithms that can be used. This design does have a flaw though, which is for the third step, every number in the matrix must be used; if they are not all used, it might be possible for two inputs to be similar enough that the differences between them don’t affect the numbers which are used to create the digest, therefore both creating the same output, resulting in a collision, and jeopardizing the security of the algorithm. The reason for choosing size 32x32 specifically is that for the third step, I wanted to use a strategy where it gets split into sixteen 8x8 submatrices, which each are used to create 2 digits of the message digest. All things considered, I really like the idea of the RNG Matrix, and I think it has a lot of potential for future applications, even if this cryptographic hash function is found to be insecure.

When it comes to the values assigned, RC4 is shockingly simple, as it only consists of the numbers 0 to 255. During my research, I found this to be a curious design, and I don’t exactly understand the decision for a pseudorandom number generator to start with something so uniform. For my cryptographic hash function, I decided it would be better to start with slightly more random numbers, and this should theoretically create a more random output. The algorithm starts by looping through each character, and does its best to create a stream of pseudorandom numbers from each one. Each character essentially creates its own RNG Matrix of numbers, and they all get condensed into a single matrix by doing a bitwise XOR operation on every number in any given location. After it is complete, the RNG Matrix is initialized with 1024 pseudorandom numbers. Compared to RC4, this should generate much more consistently random outputs, though it has yet to be tested.

Step two of the CHF, which swaps all of the matrix value locations, is the most similar part of the algorithm to its RC4 equivalent, the key scheduling algorithm (KSA). The KSA, in a nutshell, works by looping through all the values in the array S, gets a location, and swaps the two value locations. The KSA gets this location from a mixture of data from the array S and the key, which helps to randomize to values a bit more. The code behind it is quite simple, but the cryptographic hash function actually has code simpler than RC4 for this step, as it only gets the location of the value to swap with from the RNG Matrix. Since the matrix already has pseudorandom numbers in it, there is no need for the extra level of randomness by getting data from the input. In fact, the only reason why this step is necessary at all is that there may some residual patterns left over from the pseudorandom number generation that haven’t been accounted for.

The third and final step of the cryptographic hash function, the digest creation, is very different from the pseudo-random generation algorithm (PRGA) from RC4. The most notable difference between the two is that the PRGA keeps swapping more values like in the KSA while it is creating more random numbers, while the CHF only condenses the present values into a 32 digit digest. For RC4 this is necessary, since it has to be able to create a stream of random numbers indefinitely, but for the CHF, it is only required to create a 32 digit output. As stated before, how the third step works in the cryptographic hash function is it splits the 32x32 matrix into 16 separate 8x8 matrices, and from each of those it creates two digits of the message digest. There are many other ways that this information condensation step could be conducted, but I wanted to stick with something a bit simpler. One of my earlier ideas involved using a pathfinding algorithm to traverse the matrix and create the digest as it continued; however, I couldn’t come up with a way to implement this in a way that would always use every number, which as stated before, must be done on this step. This versatility is my favorite aspect of the RNG Matrix, since the number of ways you could implement this third step is virtually limitless, and this is just a single way that it can be used.

Algorithm Framework

Now that a broad overview of that algorithm has been covered, as well as its relationship with the RC4 stream cipher, we can delve deeper into how this algorithm functions. Each of the three parts started with a very simple concept, where each of the parts were very similar to RC4. However, as I began writing the cryptographic hash function, each part quickly became more complex as it evolved to become a more robust algorithm. For some of the parts there were many different iterations, but they always stayed within the planned three step process adopted from RC4. This section will cover each part in detail including explaining the pseudocode, and any changes that were made as development continued.

Part 1: RNG Matrix Initialization

Just with a first look at the cryptographic hash function’s pseudocode for the first part (see fig. 2), it is immediately evident that it is much more complex than RC4. However, once you look at it more closely, it in reality is not that hard to understand. It is comprised of two main for loops, the first of which simply initializes every element in the RNG Matrix to 1. The second main for loop is where all of the math is, and it begins by looping through every character in the input. Inside this for loop, there are two more nested for loops that traverse every location in the RNG matrix. After doing some calculations, for every location in the matrix, the algorithm does a bitwise XOR operation on values generated from every character in the input.

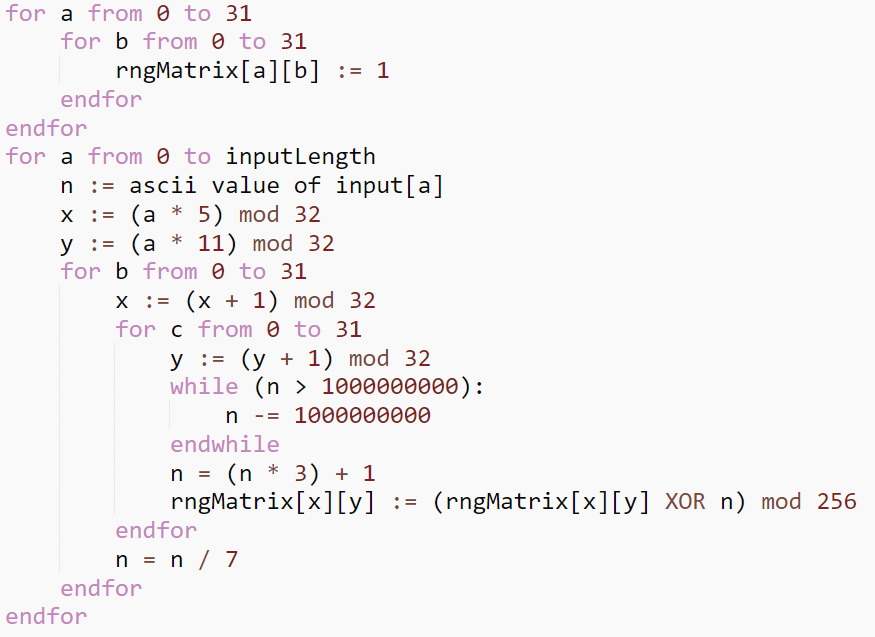


Fig. 2. Pseudocode for part 1 of the cryptographic hash function, which creates the initial

values for the RNG Matrix.

You might be wondering why the decision was made to start each value initially at one, instead of the regular zeros. This is because the original plan was to multiply every value by each other, instead of doing a bitwise XOR. But I quickly discovered that using multiplication would result in almost every number becoming zero, so the change was quickly made; this is just left over from the initial plan, and I haven’t had the time to find a more optimal alternative.

Within the second major for loop, there are six separate variables, but the only very important ones are x, y, and the internal value n. the variable a is only used to initialize these important variables, while b and c aren’t used outside of their respective for loops at all. The variables x and y are used to offset the starting location for each character, regardless of what it is. This helps break up many of the patterns created, while also making the avalanche effect much more effective, as swapping two characters will drastically change the values that are created. This also helps make longer inputs much more secure, as over time the starting points will evolve as they go from (0, 0) to (5, 11), (10, 22), (15, 1), (20, 12), etc.

Internal value n is by far the most important though, as it is used to generate the RNG Matrix values. It begins as the ascii value for a character, and morphs and evolves over the 1024 iterations as it continues generating new values. The line “n = (n \* 3) + 1”, what I call the seed function, does most of the heavy lifting for the pseudorandom value generation. I used the function 3n + 1 for the seed function since it is simple yet reliable, but it does have its fair share of issues that the other statements make up for. Since the seed function only increases in value, the while loop immediately before it help to regulate the value, ensuring it stays in bounds of the unsigned integer value limit. As a side effect, it also eradicates any patterns created by 3n + 1 that are of length 32 or shorter. However, there are still larger patterns that occur, usually around length 128. That is why the line “n = n / 7” exists; it runs after every line of the RNG Matrix is generated, and it destroys any patterns that still exist of length 32 or higher. With these safeguards in place, part 1 of the cryptographic hash function in practice produces very little trace of patterns, even with inputs of a single character. I do a demonstration of the outputs from this part in the presentation video created for this project (Ingham).

Part 2: RNG Matrix Value Swapping

Of all three sections of the cryptographic hash function, this one remained most similar to its RC4 counterpart, the KSA. As you can see from the pseudocode (see fig. 3), it’s very simple; all it does is loop through all the elements of array S, calculate a location, and swaps two values. The KSA gets this location based on information from the variable j, which changes over time much like internal value n from part 1, the value at the location that will be swapped, and the key. Since RC4 doesn’t start with pseudorandom numbers like the cryptographic hash function does, this part provides most of the randomness for the algorithm. The CHF I designed has almost identical pseudocode, with a few major differences.

Text, letter

Description automatically generated

Fig. 3. Pseudocode for the key scheduling algorithm (KSA) from the stream cipher RC4. Since

part 2 of the CHF is based on it, they share many similarities. Image source: [Wikipedia.org](https://en.wikipedia.org/wiki/RC4)

The main difference between the cryptographic hash function’s second part and the KSA is how the swap location is generated. As stated earlier, the CHF already contains pseudorandom numbers, so obtaining information from the input is just not necessary, although it’s possible it might improve the algorithm, given more time for testing and experimentation. In the pseudocode (see fig. 4), the CHF generates the x and y coordinates for the swap location solely on the current x and y coordinates, and the value located at that point in the RNG Matrix. Even though it is very similar to the KSA, it has a separate purpose; where the KSA exists to randomize the contents of array S, the cryptographic hash function uses it to stamp out any residual patterns left over after part one has been completed. This is also demonstrated in the presentation video (Ingham), and it seems to serve its purpose very well, although there hasn’t been as much testing on it compared to the other two parts.

Text

Description automatically generated

Fig. 4. Pseudocode for part 2 of the cryptographic hash function, which swaps values

pseudo randomly, depending on the value and position of the value in the RNG Matrix.

Part 3: Message Digest Creation

Something that’s very obvious with this section is just how different it is from its RC4 equivalent, the pseudorandom generation algorithm (PRGA). This is because of a fundamental difference in the output between the two algorithms; RC4 is designed to output a stream of random bits indefinitely, while the cryptographic hash function outputs a 128 bit message digest. As stated before, the third part of the CHF functions by splitting up the RNG Matrix into sixteen 8x8 submatrices, called chunks, which are each tasked with generating eight bits of the output. As you can see from the pseudocode (see fig. 5), this section is made up for four nested for loops: two for traversing between all the separate chunks, two for traversing all the values contained within any given chunk. After each chunk has finished generating two digits of the output, it appends them onto the end of the message digest.

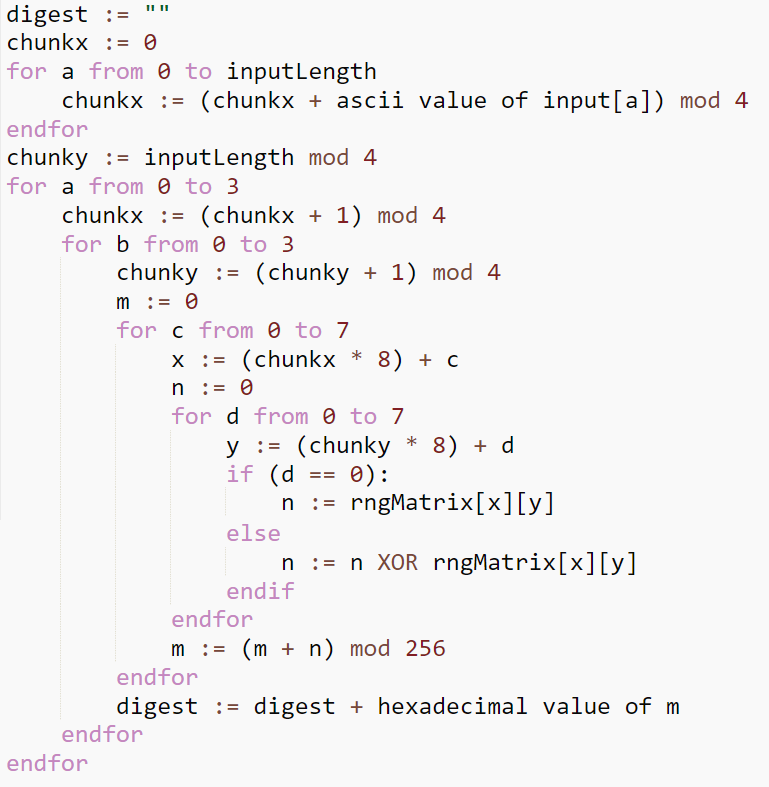


Fig. 5. Pseudocode for part 3 of the cryptographic hash function, responsible for condensing

the entire RNG Matrix into a single 32 digit (128 bit) hexadecimal message digest.

Comparing this section to the first part of the cryptographic hash function, it is much more complex and contains more variables, which are all important and have their own individual purpose. Chunkx and chunky are used to offset the variables a and b so that the algorithm doesn’t always start in the same location. The variables a and b themselves aren’t used outside of their respective for loops, while c and d are used in conjunction with chunkx and chunky to create the variables x and y. These two are very important, and they’re used to locate the true coordinates of a value within the RNG Matrix. Finally, we have the variables used to condense information, with n used to condense every value in a chunk line into a single number, and m used to condense every n value in a chunk into a single number.

The variables chunkx and chunky are very similar to x and y from the first part of the CHF, as both are used to offset starting locations within the RNG Matrix. Chunkx and chunky are initialized with data taken from the input; chunkx is the sum of all the ascii values in the input, while chunky is the length of the input. Moving further into the algorithm, n is initialized simply as the first value in a chunk line, where it later becomes the result of a bitwise XOR operation between it and all the other values in the line. Once all the n variables are finished being computed, m becomes the sum of all of them, then takes the modulus of that value to keep it under 256 (in bounds for an 8 bit value). Then after the m value is found, it gets converted to hexadecimal, and appended to the digest. Using this strategy, the cryptographic hash function creates a digest with negligible bias for any character, and is satisfyingly random.

Algorithm Testing and Optimization

Explaining how a new cryptographic hash function works is great, but if it doesn’t work well in practice, then the previous dozen pages would be completely pointless. And that’s what this section is all about; I conducted three tests for the CHF I created, and here I will detail the results of them. The first test was for the algorithm speed. Not only is this test very important for a CHF, as explained in the cryptographic hash function analysis section, but it is also very easy to implement and test. The second test, the message digest character bias test, analyzes many different message digests from a file, and finds if there is a major character bias in any of the digits. Finally, the third tests the avalanche effect, and uncovers how likely a message digest is to change if the input is very slightly changed. There were many more tests I wanted to do, such as a test for detecting any collisions, however due to time constraints, I was not able to implement this.

Part 1: Algorithm Speed test

This test, in theory, is very simple: the faster the algorithm runs, the more effective it is. As explained in the cryptographic hash function analysis section, speed is very important for CHFs to exhibit since they are required to be able to be ran on any device, including very old and underpowered ones. It is also, comparatively speaking, very easy to create a secure CHF with no time limit; finding a good balance between speed and security is what makes designing a cryptographic hash function so tough. This test was very simple to implement, just measuring the time before and after running the algorithm, then subtracting the first time from the second to calculate the time elapsed, which is usually in the realm of microseconds. Using this data, we can theoretically compare the CHF with others to see how fast it is.

One problem with this test is the fact that different lengths of inputs can greatly affect the algorithm runtime. For example, running the CHF with a single character input usually takes about 1250 microseconds. But because of how part 1 of the algorithm is designed, adding more characters to the input greatly increases the time required, to the point where a 100 character input takes over ten times as long to compute. To help standardize this test, I wrote a test function that chooses inputs randomly of lengths from five to twenty characters long. After running this function for 100 thousand randomized inputs, the average runtime always comes out at around 1100 microseconds, 150 microseconds faster than the single test with a single character. With no differences between the timers for the single test and multiple input test functions, this leads me to believe that it takes longer to run the algorithm for the first time, and all subsequent runtimes will be much faster. I would like to investigate this further, but due to time constraints this test will have to remain inconclusive and, quite frankly, very unhelpful for the task of measuring the speed of the cryptographic hash function.

Part 2: Message Digest Character Bias Test

Another aspect of CHFs which is very important is having the message digest be reliably random. It’s something that, unlike runtime speed, is very hard to observe with the data that is given. An ideal cryptographic hash function has every character being equally likely to appear for any given input, with no biases for any characters at any locations in the digest. By analyzing thousands of different message digests, a trend may appear so that we may more easily observe if there is any character bias for the CHF. If there is a certain digit that is more likely to appear in a spot in the digest, this may mean that the algorithm is insecure.

For implementing this test, I made use of the digests created with the multiple input test described earlier for measuring speed by saving them to a file. Then, later we can read the digests from the file, and count every character and every location. With this data, we can organize into counts for how many times each character shows up at each location, then convert those counts into percentages. The optimal output, since there are sixteen different possible characters in a hexadecimal digest, is for every percentage to be 6.25%. After running this test with digests from 100 thousand random inputs, I found that the 6th character in fact has a 7% chance to be a zero, 0.75% higher than it should be. This bias is small enough that it may be negligible, but it does show that the function is not perfectly random, and it may have a vulnerability that can be exploited.

Part 3: Avalanche Effect Test

The final test I ran against the cryptographic hash function is my personal favorite, the avalanche effect test. How it works is the algorithm is run on a randomized input, after which the input is changed very slightly, and the algorithm is run again. Every time the input changes, one option is randomly chosen out of three: adding a random character, deleting a random character, or swapping two random characters. After the algorithm is ran for both inputs, the message digests are then compared; it counts every time a character changed, then converts this count to a percentage. After many iterations, it finds the average percentage, and this determines the avalanche effectiveness.

I decided to run the algorithm with ten thousand iterations, and even after repeating the test multiple times, it always returned with an avalanche effectiveness of 93.4%. What this means is that when slightly changing the input, each character has a 93.4% chance of changing; this sounds like a very suboptimal result, but it makes sense upon further inspection. In an ideal cryptographic hash function, every time the input is changed, every character has a 6.25% chance of appearing in any given location. This also means that there is a 6.25% chance that it may be the same character, which is also the same chance for two inputs that are completely unrelated. With this factored into the test, the theoretical maximum avalanche effectiveness is 93.75%, just 0.3% away from the results obtained from the CHF. These results mirror the ones from the message digest character bias test: the cryptographic hash function seems to be near perfect, but just barely falls short.

Conclusion

With all the tests concluded, they all (with the exception of the inconclusive speed test) seem to suggest that the cryptographic hash function I designed is relatively strong and secure. However, we can’t come to that conclusion without more tests being run, which is something that this project just didn’t have enough time to implement. There are so many more tests I was planning to conduct, so that I could measure the CHF’s resistance to collision, pre-image attacks, and second pre-image attacks. It’s possible these tests could reveal an undiscovered major weakness that could severely undermine its security. Regardless, there is still the fact that the cryptographic hash function performed very well in the tests that were implemented, and until more tests are run, we can assume that the algorithm is secure and robust.

However, even if the CHF is found to be flawed, the RNG Matrix in concept is a very strong tool that could be used for other algorithms. It is a very versatile design, with so many different ways that the values could be initialized, and just as many ways that the values could be condensed to create pseudorandom numbers. Even if this implementation of the RNG Matrix is suboptimal, there is likely an ideal solution that creates perfectly random numbers. It is for this reason that even if the cryptographic hash function I designed is severely flawed and insecure, I consider this project a success.

Works cited

Das, Suman, et al. “An Approach of Refining RC4 with Performance Analysis on New Variants.” Sadhana, vol. 44, no. 11, Nov. 2019, p. N.PAG. EBSCOhost, [doi:10.1007/s12046-019-1209-7](http://ezproxy.uhd.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=140156237&site=ehost-live).

Ingham, Monica. “Senior Project Presentation - Monica Ingham.” YouTube, 13 Aug. 2021, [youtu.be/aXl-BmkRgt4](https://www.youtube.com/watch?v=aXl-BmkRgt4).

Kishore, Neha, and Priya Raina. “Parallel Cryptographic Hashing: Developments in the Last 25 Years.” Cryptologia, vol. 43, no. 6, Nov. 2019, pp. 504–535. EBSCOhost, [doi:10.1080/01611194.2019.1609130](http://ezproxy.uhd.edu/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=138886234&site=ehost-live).

Wikipedia, “Cryptographic Hash Function.” Wikipedia, Wikimedia Foundation, 6 July 2021, [en.wikipedia.org/wiki/Cryptographic\_hash\_function](https://en.wikipedia.org/wiki/Cryptographic_hash_function).

Whittaker, Zack. “Md5 Password Scrambler 'No Longer Safe'.” ZDNet, ZDNet, 7 June 2012, [www.zdnet.com/article/md5-password-scrambler-no-longer-safe/](http://www.zdnet.com/article/md5-password-scrambler-no-longer-safe/).