Simplified Blockchain

Author: Arash Dehghan
Supervisor: Dr. Angèle Hamel

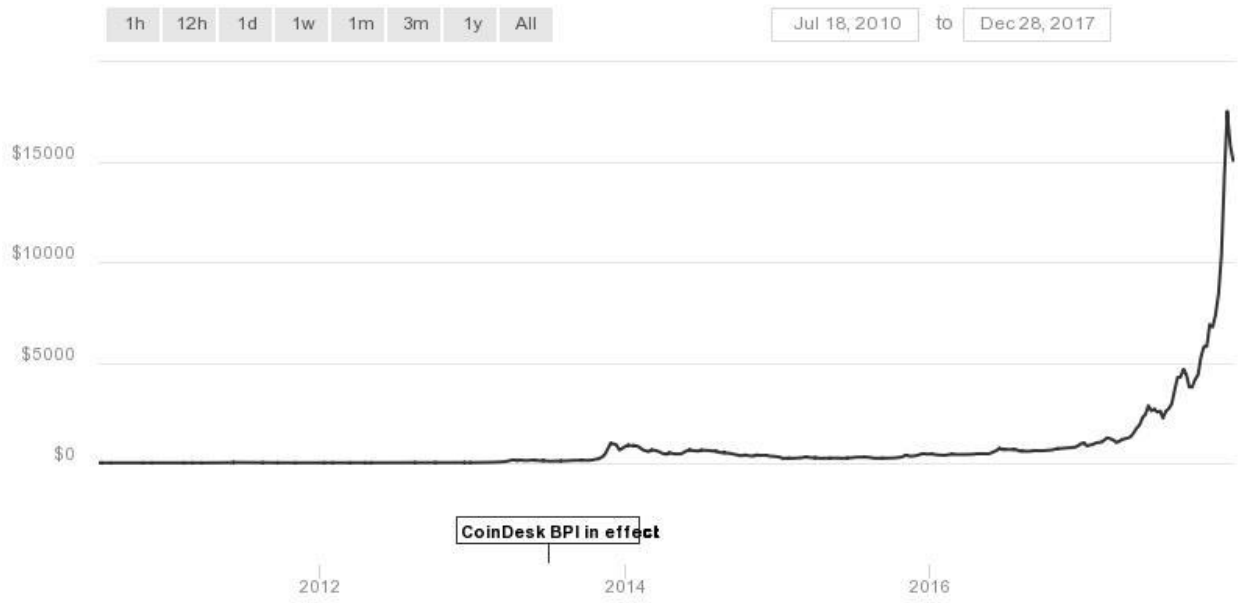
Introduction and Abstract

In November and December of 2017, the cryptocurrency known as Bitcoin soared to record-high values, reaching at one point a worth of nearly $20,000/1BTC[1]. People began investing heavily in Bitcoin and other cryptocurrencies alike, creating a global craze. With this recent success, many questions have been brought to the forefront regarding its security. Bitcoin's security is cemented through its application of Blockchain, which centres around the utilization of the SHA-256 hashing function. The SHA-256 hash function, although secure in practice, is difficult to comprehend in a step-by-step manner due to its complexity. This paper looks to construct a simplified hash function, operating on 64-bits rather than 256-bits, and to run it through necessary statistical testing to ensure its relative security. Thus, we will look to downscale the set SHA-256 algorithm while still maintaining the functions overall structure. By doing so, we aspire to create an easier to understand version of the SHA-256 function, making it more accessible to the average novice. This kind of downscaling has been previously done for AES (Advanced Encryption Standard) and DES (Data Encryption Standard) and this paper will look to build on that work. This work is relevant, as by understanding the inner-workings of SHA-256 and its features, we will be able to develop more secure hash functions in the future. In the first section of this thesis, we will look at Bitcoin and Blockchain, discussing their structural characteristics. We will also introduce hashing and explore the different avenues in which it is used. In the second section, we will introduce our simplified 64-bit algorithm, and go through an example of the down-scaled SHA-256 function. In the third section, we will place our function through statistical testing, running said function through tests such as the Strict Avalanche Test (SAC) and Collision Test. In section 4, we will summarize our findings and making our final conclusions.

Section 1: Background Information

Bitcoin

Bitcoin is a worldwide cryptocurrency and peer-to-peer digital payment system, originating in January of 2009, created by an anonymous individual/group of individuals going by the name of Satoshi Nakamoto[2]. Bitcoin has seen a large spike in its value in the final months of 2017, having its value sky-rocket by thousands of dollars per day, and even per hour. Below is a chart depicting Bitcoin's value since July of 2010:



Via: Coindesk.com/price/

Bitcoin contains a number of characteristics which distinguish it from normal currencies:

1. Bitcoin is decentralized, meaning that there is no central administrator that regulates it (i.e. a bank or government)[3]. It is community regulated and operated, meaning also that there is no 1-800 number to call if anything negative transpires.
2. Bitcoin is similar to cash, in that unlike other forms of digital payment, it is untraceable. This means that individuals conduct transactions anonymously[3].
3. Instead of using personal information to identify an individual, such as a birth name, address, etc., Bitcoin uses public and private keys as identifiers[3]. Public keys are large numerical values which are used to encrypt data whereas private keys are large numerical values that are mathematically linked to the public keys[4]. In asymmetric cryptography, whatever is encrypted with a public key may only be decrypted by its corresponding private key and vice versa[5]. You can think of this similar to your cellphone number and the password to your phone. Although your phone number is public information, as is your public key, only you know the password to unlock your phone. Similarly, only you have the private key which unlocks the information sent to the corresponding public key.

Bitcoins security comes from its implementation of Blockchain, and in turn, hashing. We explore both Blockchain and hashing in the next section.

Hashing

Hashing is the transformation of a string of characters into a fixed-length value or key that represents the original string[6]. In other words, hashing is used to create a signature of data, similar to a fingerprint. All hash functions share several key characteristics which are listed below[4]:

- Distinctness: Each hash value is ideally unique to its input, meaning that there is a one-to-one mapping between the input and the hash-value (Although we will see this is impossible in practice).
- Presumed Randomness: Any small change in the input will result in a drastically different hash value.
- One-Way-ness: Hash functions are one-way functions, meaning that once an individual has received a hash-value for a specific input, they are not able to retrieve the initial input back from the hash-value.

- Fixed Length: Regardless of the size of the input, the hash function will produce a hash value with a fixed length (The SHA-256 hash will be 64 Hexadecimals (or 256-bits)).

In addition to its use in Blockchain technology (and thus Bitcoin), hashing is also used in areas such as password encryption and file signature verification. In an attempt to better understand hashing, let us first consider its use in a field which is familiar: password encryption. Consider an instance in the past when you have forgotten your password to a specific website. You may have noticed that when reporting to said website your forgotten password, the website requires that you change your password, rather than giving you your password back after a number of security questions. This is due to the fact that the website does not actually know your password, since it does not store your password in their databases as your actual password, but rather as a hash value of your password. Once you submit your password for the first time, that value gets hashed and stored into the databases while your actual input is discarded. As we have already discussed, since hash values have a one-way characteristic, the website would not be able to retrieve your original password from their stored hash value of it. This is done by institutions to secure their users data in the case of a break in, where a hacker would only be able to obtain a hash value of an individual's data. This is again one of the reasons people are told to set their passwords as something that cannot be brute-forced, such as '12345', or their name, etc., since a hacker would be able to guess all of this data and compare it to their stolen hash-value of your actual password.

Below are several examples of different inputs, both as passwords and not, and their resulting hash-values when put through the SHA-256 hash function.

Example 1.1

Password: FlyGuy33 → 2401e12bc25eb646f152e41e5705df8b4dab70a237a7be80a3455ba606340bd6

Password: flyGuy33 → fdf68394ba299d2370aec4d5f8985537d08ca34b49ddb7d161d00fb09b693928

Script of *The Notebook* → 1602bd22aa44c35047147f615b95872c2ff5b1312db27e554bd07ddf4fd2f583

"" → e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

We see when comparing the first two passwords that although two inputs may be nearly identical, any small change will alter their hash values drastically. In our case, the only element changed is the lowercase and uppercase 'F's' at the start of our password. Thus, we derive the characteristic of presumed randomness. Next, we compare the two subsequent examples. First, we have the SHA-256 hash-value of the entire script of the film *The Notebook*, which consists of hundreds of thousands of characters of information, all of which are compressed into a 256-bit hash-value. Next, we consider the SHA-256 hash-value of an empty string. Analyzing this, we can see that regardless of whether we're inputting an empty string, or the complete script of *The Notebook*, the result of our hash-value is the same fixed length, fulfilling the fixed length condition. Finally, if we compare all four examples given, we see that all hash-values are distinct.

Collision Resistance in Hash Functions

A cryptographic hash function is collision resistant if it is difficult to find two inputs that will result in the same hash value[7]. Once a hash function (such as SHA-1) has been broken, meaning a collision has been found, the community progresses to using more secure hash functions, such as SHA-2 or SHA-3. The cryptographic community is always working on developing more secure and collision resistant algorithms to keep up with the growing strength of computing power. Thus, it is important when constructing and/or analyzing hash functions to recognize what characteristics help them become more collision resistant. As we progress through this paper, we will point out certain traits which help ensure this security.

Next, we will define Blockchain and discuss how it operates. Through this process we will see how hashing, and in specific SHA-256, is utilized within the Blockchain.

Blockchain

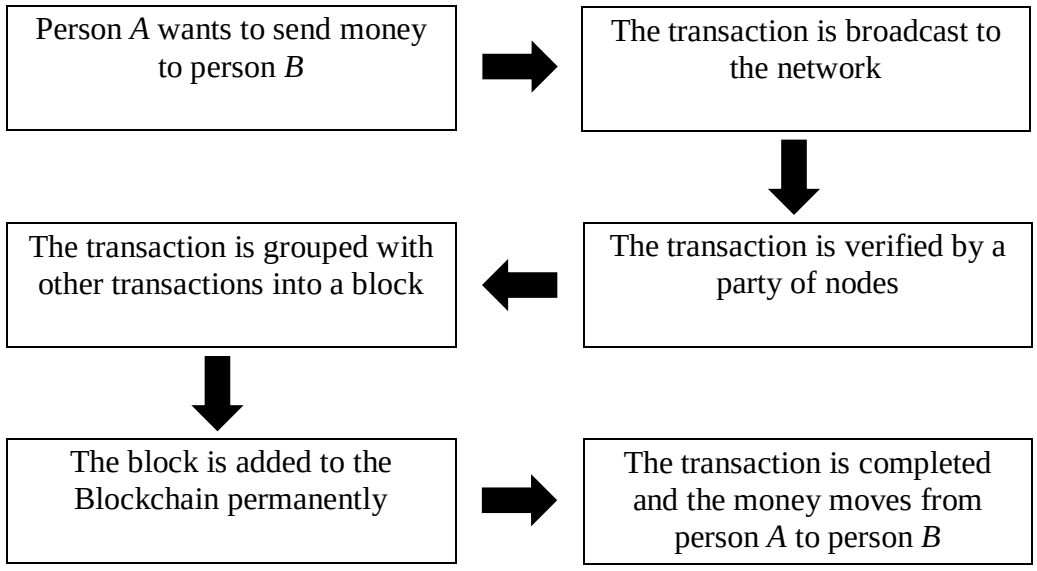
A *Blockchain* is a digital ledger in which transactions made in bitcoin or other data types are recorded chronologically and publicly. Bitcoin's use of Blockchain can be classified as being one of the key reasons behind its vast success. Blockchain's binding of transaction history into a straightforward public ledger has created a great deal of interest in other markets ranging from

supermarket manufacturing to financial banking. This kind of digital ledger has a great number of benefits which distinguish it from standard monetary systems. These benefits include[4]:

- Decentralization: The use of a consensus mechanism over an intermediary third-party validation system to keep track of transactions (i.e. non-reliance on banks or governments).
- Anonymousness: Confidentiality of user information through the use of public and private keys as opposed to personal user data such as name, address, etc.
- Cost Efficiency: The elimination of third-party transaction fees (i.e. bank fees).
- Transparency: The ability for the public to have access to the entire blockchain and the contents of its ledger.
- Security: The encryption of all transactions on the blockchain through cryptographic hash functions.
- Immutability: The inability of an individual to alter a block once it has been added to the Blockchain (i.e. a fixed ledger).
- Efficiency: The improved speed of transactions due to the Blockchain's avoidance of a lengthy verification, reconciliation, and clearance process.

Below is a visual representation of how Blockchain operates in Bitcoin:

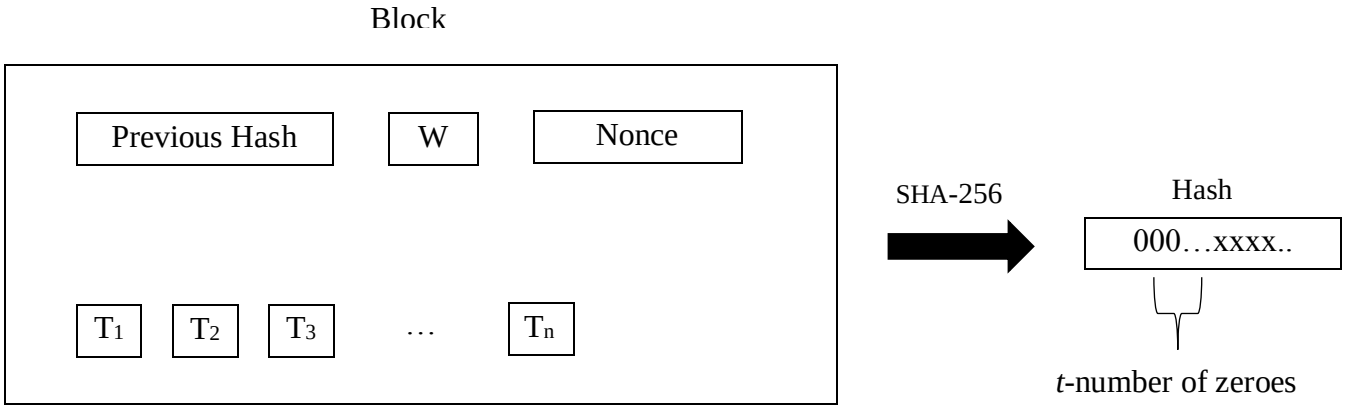
Figure 1.1



We can see from Figure 1.1 that once a transaction is broadcast to the network, it must be verified by a group of nodes, where a node is defined as being any computer that connects to the Bitcoin network. A transaction is verified upon its passing of a set of rules in the Bitcoin protocol.

Nodes, for example, must verify that the inputs of the transaction information are valid, that the coin(s) being spent hasn't been double-spent, that the output and input are equal. A miner (i.e. an individual adding the block to the Blockchain) can break these rules while mining the block (adding the block to the Blockchain), but his/her block will not be accepted by the community and he/she will not receive the monetary reward that comes from mining the block. This would incentivize miners to follow said rules when adding blocks to the Blockchain. Each block in the Blockchain, along with containing a group of recent transactions, is comprised of the hash value of the previous block, the public key of the miner creating the block, and a nonce (defined below). All of this is put through the SHA-256 hash function to produce a hash value assigned to the specific block. Below is a visual representation of this:

Figure 1.2



W: Public Key of the miner that is creating the block
$T_1 \ldots T_n$: A collection of transactions over a fixed period of time (Ex: 15 minutes)
Nonce: A general string which is used once (Ex: The date and time)

As we can see from Figure 1.2, the block being mined, along with all of the information within it, is put into a general format and processed through the SHA-256 hash function, which will produce a 256-bit hash value. This hash value will act as a fingerprint for said block, and as we have discussed, any small change in either the previous hash, the public key, the nonce, or the transactions, will alter said value drastically. To have his/her block added to the Blockchain, the job of the miner is to produce a hash value of said block that begins with *t*-number (in bits) of zeroes, where *t* is an ever-changing integer (As of writing this, the value of *t* is 18)[3]. However, it is important to note that the only thing a miner is able to alter in a block is the nonce, in which the miner would slowly increment the nonce bit-by-bit until the desired hash value is achieved. By imposing this restriction on *t*, the system forces miners to devote their time, money (in terms of electricity), and energy into completing this task and discourages bots and/or just anyone from reaping the monetary rewards. This, along with community verification, prevents any false transactions from seeping through into the Blockchain. By doing this, the system creates a lottery system, where regardless of the power of your computer, your chances of producing the desired hash value are no greater than anyone else's. Let's consider the scenario where each individual

starts with an identical block, and increments their nonce until they get the desired hash value. In this case, since the starting block for each miner is identical, the miner with the most powerful computer would win every time. However, in reality, each block contains a public key unique to the miner, and a unique nonce. Each individual miner starts with different information in their block, and thus would have theoretically the same odds of winning the race to mine the block as everyone else. Bitcoin protocol attempts to maintain a ratio of one block being mined every 10 minutes[3]. To accomplish this, the system automatically adjusts the difficulty of mining a block every two weeks (or 2016 blocks)[3]. If the 2016 blocks are found too easily, the value of *t* would increase in our desired hash value. Thus, the difficulty of mining a block is set by the protocol to be proportional to the amount of work being put in by the community. To better understand all of this, let's consider an example.

Example 1.3

Let's assume that Roger would like to mine a block to the Blockchain. He would first gather up all the necessary information (i.e. the transactions he would like to include in his block, his public key, etc.), and would set a nonce, let's say to that day's date. Then, Roger would put all of the gathered information through the SHA-256 algorithm and have a hash value returned to him. If this hash value started with the desired number of 0's, he would broadcast his block to the network, letting the community know that he had found a satisfactory hash value. The community would then verify that the inputted information would result in the desired hash value, and the block would be added to the Blockchain. However, if Roger does not get the desired hash value on his first attempt, he would then increment the nonce and try again. By this, we mean that he would perhaps change a 0-bit in his nonce to a 1-bit (or vice versa), and so forth. He would continue this process until either he, or someone else, had successfully mined said block (i.e.: gotten the correct number of leading 0's). As we discussed before, Roger could try to include an invalid transaction into his block, but this transaction would be flagged by the community, and his block would not be added to the Blockchain. Again, since every miner starts out with a different public key and nonce in their initial block, each individual would have the same odds of mining a specific block. This process would be similar to a lottery, where having one computer mining the block would be equivalent to having one lottery ticket. Thus, even if Roger convinced ten of his friends to also attempt to mine the block, their odds of winning would still be relatively low. Now, let's assume Roger attempted to alter a transaction in a block, say block *A,* that was already mined to the Blockchain. For example, let's say that Roger wanted to alter a transaction that had taken place between himself and his friend Don, where the original transaction consisted of Don sending Roger 1 Bitcoin, and Roger attempted to modify this transaction to say that Don had instead sent Roger 1000 Bitcoins. By changing this one transaction in block *A*, the entire hash-value of said block would be altered and would thus not connect to the block ahead of it, block *B*, which used block A's original hash value. This, again, is because each input into the SHA-256 hash function would produce a unique hash value. Thus, the Blockchain would be broken and the altered block would not be accepted.

Now that we understand how hashing and Blockchain work, we can see that the Blockchain, and in turn Bitcoin, is heavily reliant on the SHA-256 hash function, which connects each block in the

Blockchain and prevents fraudulent activity from taking place by ensuring a unique hash-value for each input. This paper will now explore the inner-workings of this SHA-256 hash function and will attempt to better understand the characteristics which guarantee its security by down-scaling it and making it more manageable.

Section 2: Simplified Blockchain

The goal in this section is to implement the SHA-256 hash function on a smaller scale while still maintaining its overall algorithmic structure. There are a number of elements of the SHA-256 function which we will modify, and others which we will leave unchanged. These elements are listed below:

Modified Elements:

- Utilization of 8-bit hash values rather than 32-bit hash values
- Conversion of chunk sizes from 512-bits to 128-bits
- Adaptation of bitwise operations
- Reduction of final hash value from 256-bits to 64-bits

Unchanged Elements:

- The number of hash values and round constants
- The number of rounds in the schedule array and the compression function

Once a simplified version of the SHA-256 hash function is constructed, we explore its security using several different statistical tools. The results from the security testing will be covered in section 3.

Implementing the SHA-256 Algorithm on a 64-bit Hash Function

We will begin by defining an overview of the steps taking place in the algorithm.

Overview

1. Initializing Hash Values and Round Constants
2. Pre-Processing
3. Processing Message (For each 128-bit chunk)
    a) Initializing Working Variables
    b) Compression Function
    c) Adding Compressed Chunk
4. Producing Final 64-bit Hash Value

Step 1: Initializing Hash Values and Round Constants

To initialize our hash values, we follow the algorithm of:

1) Take the first 8 prime numbers: 2,3,5,…,19.
2) For each prime number, take the square root, then take the decimal portion.
3) Multiply your answer by 2^8, and take the integer portion of your answer.
4) Convert your answer into hexadecimal (for convenience).

Example 2.1

Let us follow this algorithm for the prime number *2.*

1) $\sqrt{2}$ = 1.414213…
2) Take the decimal portion: *A* = 0.414213
3) Multiply *A* by 2^8: (*A*) x (2^8) = 106.03867
4) Take the integer portion: 106
5) Turn 106 into hexadecimal: H_0=6A

Repeating this process for the rest of the first 8 prime numbers, we arrive at our hash values, which are presented below:

H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
6A	BB	3C	A5	51	9B	1F	5B

To initialize our array of round constants, we follow an almost identical algorithm:

1) Take the first 64 prime numbers: 2,3,5,…,311.
2) For each prime number, take cube root, then take the decimal portion.
3) Multiply your answer by 2^8, and take the integer portion of your answer.
4) Convert your answer into hexadecimal

Following this process for the first 64 prime numbers, we arrive at our round constants, which are presented below by initializing our array K[i], where $0 \leq i \leq 63$:

$K[0\ldots63]=$

0	1	2	3	…	62	63
42	71	B5	E9	…	BE	C6

Remark 2.1

It is important to note that we have modified our initial hash values and round constants in this step from 32-bits in the regular SHA-256 function to now 8-bits in our new 64-bit hash function. We have done this by multiplying the decimal portion of our hash values and round constants by 2^8 rather than 2^{32}. Furthermore, we use prime numbers to initialize both portions for two reasons. First, all square and cube roots of prime numbers are interesting and unique, making their decimal portions useful in the procedure. Secondly, we want to pick values which are well

defined and are open to the public. We do not want to select a random set of private numbers that only a select number of individuals know (Ex: CIA), allowing them to have back-door access. With regards to our array K, we decide to maintain the 64-entry array length, as to maintain the structure of the SHA-256 function.

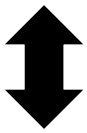
Step 2: Pre-Processing

To begin pre-processing, we must first select a string/input we want to have processed. Let's say that for this example we use the string:

"I love Bitcoin!"

To create a 64-bit hash value of this string, we must first begin by converting it into binary (refer to Appendix A):

"I love Bitcoin!"



01001001001000000110110001101111011101100110010100100000010000100110100101110100011000110110111101101001011011100010000

Now, we will find the length of this binary string, setting the value equal to *L*. In this case, we have that *L*=120. Next, we will append a single '*1*' bit to the end of the message:

01001001001000000110110001101111011101100110010100100000010000100110100101110100011000110110111101101001011011100010000<u>1</u>

Now, we will append *K '0'* bits to the end of our string, such that *L*+1+*K*+64 is a multiple of 128 (We will see why it must be a multiple of 128 later on). Thus, in this case, we append K=71. Thus, we have the resulting 256-bit (256=128x2) binary string:

01001001001000000110110001101111011101100110010100100000010000100110100101110100011000110110111101101001011011100010000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Where we append 64 bits at the end of our string, denoted by x's, to represent our yet to be determined 64-bit big-endian integer, which we will solve for next.

Definition 2.1

Endianness, in specific, Big-endian and little-endian, are terms that describe the order in which a sequence of bytes is stored in computer memory. Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address)[4].

Example 2.2

Let us place the hexadecimal value of 93 BD 16 AF into both big and little-endian:

Big-Endian	Little-Endian
93BD16AF	AF16BD93

Now, going back to our example, to determine the 64-bit big-endian integer, we must first convert *L* (which again in our case *L*=120) into binary and place it in big-endian order:

120 → *0011000100110010001100000*

Finally, for our 64-bit integer, we take our binary big-endian *L* value, and append it to 64-len(*L*) zeroes:

0000000000000000000000000000000000000000001100010011001000110000

Thus, after appending this 64-bit big-endian integer to our answer, we have our final pre-processing result:

0100100100100000011011000110111101110110011001010010000001000010011010010111
0100011000110110111101101001011011100010000110000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000001100010011001000110000

Remark 2.2

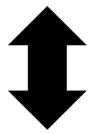
In this portion, we have altered the size chunk sizes from 512-bits to 128-bits. We will see shortly why we have chosen 128 as our new value. Also, with regards to our 64-bit big-endian integer, the largest integer value which we are able to have as our size of input, L, is 9,223,372,036,854,775,807, which translates to 64 straight '1's[8].

We are now finished with the pre-processing procedure and will move onto processing our message. In this stage, we will be utilizing the bitwise operations which are defined in Appendix B.

Step 3: Processing

To begin processing, we must first break our message into 128-bit chunks. In our case, we have two resulting chunks:

0100100100100000011011000110111101110110011001010010000001000010011010010111
0100011000110110111101101001011011100010000110000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000001100010011001000110000



(1)

0100100100100000011011000110111101110110011001010010000001000010011010010111
0100011000110110111101101001011011100010000110000000

(2)

0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000001100010011001000110000

Now, for each chunk we create a 64-entry message schedule array W[0..63] of 8-bit words and copy each chunk into the first 16 words W[0..15] of the message schedule array. In other words, we will split each 128-bit chunk into 8-bit portions, and insert each portion into the first 16 spots of our array W. We will process the first chunk to begin with, and will repeat this procedure for the second chunk later on. Thus, we create our message array for the first chunk and fill it as such:

W[0…63]=

0	1	2	…	15	16	17	…	62	63
49	20	6C	…	80			…		
01001001	00100000	01101100	…	10000000			…		

Now, to fill the rest of our array, we use the following algorithm below:

for i ***from*** 16 to 63:

S_0 = (W[i-15] ***rightrotate*** 2) **xor** (W[i-15] ***rightrotate*** 5) **xor** (W[i-15] ***rightshift*** 1)
S_1 = (W[i-2] ***rightrotate*** 4) **xor** (W[i-2] ***rightrotate*** 5) **xor** (W[i-2] ***rightshift*** 3)
W[i] = W[i-16] + S_0 +W[i-7] + S_1

W[0…63]=

0	1	2	…	15	16	17	…	62	63
49	20	6C	…	21	F5	ED	…	8F	57
01001001	00100000	01101100	…	00100001	11110101	11101101	…	10001111	01010111

Remark 2.3

It is important to note that bitwise rotations, shifts, and XOR's have been altered from the original SHA-256 algorithm. Since we have not performed any statistical testing, the new values of these bitwise operations were determined through educated guesses. For example, if in the original SHA-256 function we performed a right-rotate of 16-bits, we would now in our 64-bit hash function perform a right-rotate of 4-bits, since a rotate of 16-bits on a 32-bit hash values is equivalent to a rotate of 4-bits on 8-bit hash values. Also, since it is the case that during shifting we lose data, it is crucial to decide upon a right-shift value which maintains just enough data. By this we mean that, if we right-shift by too much in our new hash-function, say a right-shift of 7 on an 8-bit hash, we lose too much information, making the function more vulnerable to a collision. However, if we right-shift by too little, we leave our hash function susceptible to a brute force break-in. As well, we see now the reasoning behind our choosing of 128-bit chunks. By staying true to the SHA-256 structure, we have our 64-entry array, and we fill our first 16 positions in the array with our message. In the original SHA-256 function, we have 512-bits chunks since we have 32-bit hash values (i.e. 32x16=512). Now that we have 8-bit hash values, we must have 8x16=128-bit chunk sizes.

A) Initializing Working Variables to Hash Values

Next, we initialize our working variables, setting them equal to the hash values we derived Step 1. Thus, we set A through H to H_0 to H_7, respectively:

Recall the hash values we derived:

H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
6A	BB	3C	A5	51	9B	1F	5B

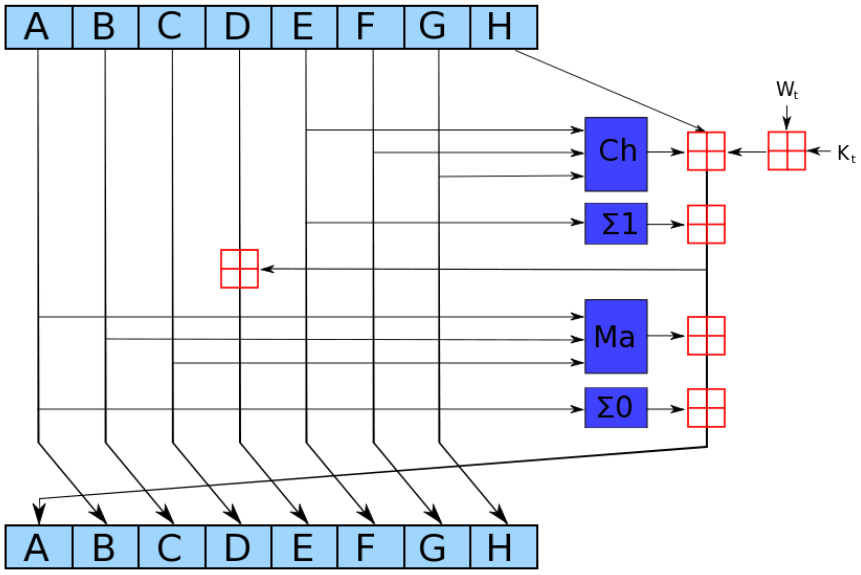
Now let:

$A = H_0$
$B = H_1$
$C = H_2$
$D = H_3$
$E = H_4$
$F = H_5$
$G = H_6$
$H = H_7$

B) Compression Function

We now arrive at the final stage of the processing procedure, where we compress each of our derived chunks into a hash value. Below is a visual representation of how the compression function operates:

Figure 2.1



Via: SHA-2 Wikipedia. https://en.wikipedia.org/wiki/SHA-2

We see from this diagram the working variables which we initialized, along with our two arrays, being put through a number of different functions. These functions are made up of the same bitwise operations we looked at earlier in the processing stage. This algorithm is better understood through its pseudocode, which is represented below:

for i ***from*** 0 to 63:

S_1 = (E ***rightrotate*** 2) ***xor*** (E ***rightrotate*** 3) ***xor*** (E ***rightrotate*** 6)
CH = (E ***and*** F) ***xor*** ((***not*** E) ***and*** G)
$Temp_1$ = H + S_1 + CH + K[i] + W[i]
S_0 = (A ***rightrotate*** 1) ***xor*** (A ***rightrotate*** 3) ***xor*** (A ***rightrotate*** 6)
Maj = (A ***and*** B) ***xor*** (A ***and*** C) ***xor*** (B ***and*** C)
$Temp_2$ = S_0 + Maj
H = G
G = F
F = E
E = D + $Temp_1$
D = C

C = B
B = A
A = $Temp_1$ + $Temp_2$

Remark 2.4

Once again, we have altered the incrementation of the bitwise operations to fit our 8-bit hash values, using the same logical estimation to pick our new values as before. We have however not modified the number of rounds in our compression function (64), as to maintain the overall structure of the SHA-256 hash function. By applying these bitwise operations to our working variables, we are scrambling our message, making it increasingly difficult for someone to retrieve our original hash values. At the end of our 64 rounds, we are left with brand new A through H working variables, which we will add to our initial hash values in the next stage.

C) Adding Compressed Chunk

Finally, we add our final working variables after the 64 rounds of compression to our initial hash values:

$H_0 = H_0 + A$
$H_1 = H_1 + B$
$H_2 = H_2 + C$
$H_3 = H_3 + D$
$H_4 = H_4 + E$
$H_5 = H_5 + F$
$H_6 = H_6 + G$
$H_7 = H_7 + H$

We now get the final hash values for our first 128-bit chunk:

H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
4C	54	12	0F	68	36	A7	B3

However, we are not yet done. We must still process our second chunk, following the same processing procedure as we just established, but using our second chunk to create our array W[0…63]. We additionally use the hash values which we just derived above as our H_0 through H_7 values, rather than the initial hash values we established in Part 1.

Overview

1. Initializing Hash Values and Round Constants ✓
2. Pre-Processing ✓
3. Processing Message (For each 128-bit chunk)
    a) Initializing Working Variables
    b) Compression Function
    c) Adding Compressed Chunk
4. Producing Final 64-bit Hash Value

Remark 2.5

Thus, regardless of the size of our initial input, we break our message into chunks and process one chunk at a time, producing a fixed length hash value every time (since each chunk is reliant on the hash values of the one before it). This again makes it difficult for someone to retrieve the original input from its hash value, as it is impossible to determine the size of the input from just its hash value.

After following this procedure for all chunks, and adding our final compressed chunk to our hash values, we arrive at our final hash values of:

H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
5A	12	D4	C0	AD	93	B3	35

Step 4: Producing Final 64-bit Hash Value

Finally, we append our final hash values together in big-endian to produce our 64-bit hash value:

H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
5A	12	D4	C0	AD	93	B3	35

"I love Bitcoin!"

5A12D4C0AD93B335

Therefore, we arrive at our final 64-bit (16 hexadecimal) hash value 5A12D4C0AD93B335 correlating to the string “I love Bitcoin!”.

Now that we have successfully scaled-down our SHA-256 hash function, we can begin running statistical testing on it to determine how good it actually is.

Section 3: Statistical Testing

According to the National Institute of Standards and Technology (NIST), which provides the standards for all algorithms in this field, there are three key characteristics which are used to define an effective cryptographic hash function. These include security, cost/performance, and algorithm/implementation characteristics. Below is a brief description of each:

1. Security

   Considering cryptographic hash functions are primarily used to store and protect sensitive data, it is easy to deduce why security is the most important factor when evaluating hash algorithms. Apart from the Blockchain, cryptographic hash functions are used in areas such as password and file encryption, where their competency is crucial to the entire system. However, as we will see, there is a great deal of debate around what security definitions should be used to evaluate hash algorithms[9].

2. Cost and Performance

   NIST identifies cost as being the second-most important criterion when evaluating hash algorithms. In this case, cost refers to the required computing memory and time necessary to compute a hash. With regards to this, it is imperative to consider an algorithms compatibility with platforms such as mobile devices, smart cards, and radio-frequency identification (RFID), where memory may be a factor. In addition, the algorithms performance (computational efficiency) on these platforms helps define its suitability and competence[9].

3. Algorithm and Implementation Characteristics

   Finally, NIST takes into account the clarity and efficiency of an algorithm. Optimally, an algorithm should be simple and elegant enough as to encourage understanding, analysis, and design confidence. In turn, it should not perform any unnecessary operations which work to slow down its run-time[9].

For the purposes of our scaled-down hash function, we will be focusing on the algorithms security and will omit the final two criteria. We do this because the SHA-256 hash function which we are scaling-down is already a standardized hashing algorithm, meaning that even by scaling-down the function, we maintain the final two elements (These components were already tested and verified

by the NIST organization for SHA-256). Rather, we will turn our attention to the security of our new 64-bit hash function and investigate how capable it is against statistical testing.

Testing for Security

As previously mentioned, there remains significant debate amongst the cryptographic community with regards to the question of how to evaluate an algorithm in terms of its security. There are hundreds of different tests which can be applied to any hash algorithm algorithm. If a hash function fails any one of the hundreds of tests, it is considered to be broken and unusable. On the other and, passing one or even all the statistical security tests does not equal complete security. Due to the restrictions of available resources (i.e.: computational power) and time, it would have been unrealistic to implement every single statistical test on our scaled-down function. A solution to this, however, was to apply the small set of tests which NIST had used in its search of the SHA-3 algorithm several years prior. In 2008, NIST began accepting submissions for a new hash function called SHA-3. Along with this announcement, NIST published a number of papers which outlined its requirements for successful candidates. In these requirements, they defined the primary qualities which they tested for when considering each algorithms security. Two of the essential qualities which they considered were the pre-image and collision resistance of a hash function. These qualities were tested using the Strict Avalanche Criterion test and the Collision Test, respectively[9]. In this paper, we will test for the same qualities using the identical tests which were used for SHA-3. (It is important to note that NIST used a large number of supplementary tests but highlighted these two tests as the essential ones in their papers). By placing our down-scaled function through these tests, we will analyze the importance of the bitwise operations in our algorithm and seek to better understand how the algorithm operates.

Pre-Image Resistance

We will begin by defining Pre-Image Resistance and discuss its importance in Blockchain.

Definition 3.1

Pre-Image Resistance looks to make it difficult, given a hash value of X, to find an input M_1 such that $H(M_1)=X$.[10] In other words, if we are given a specific hash value, we want it to be as difficult as possible for someone to be able to find an input which hashes to that value. This is relevant to the Blockchain with regards to the mining of blocks. If we recall, for a miner to be able to mine a block to the Blockchain, they must first find a hash value for their input which begins with t-number of leading zeroes (t being equal to 18 at the moment). Since there is a monetary reward for mining blocks, we want to make it as relevantly difficult as possible to arrive at the required hash value. We want the quickest way to find the desired hash value to be through brute force (i.e.: No set algorithm which provides an advantage to a given individual). In this way, every miner has an equal chance of arriving at a desired hash value (since each input is unique due to the nonce). For us to have this quality, each output-hash of our function must be random. In other words, for our cryptographic hash function to be Pre-Image Resistant, its output must be random.

Randomness in Cryptographic Hash Functions

As previously mentioned, cryptographic hash functions rely heavily on randomness. Any hash function which is considered to have its output be non-random is said to be broken, and thus, not secure and unusable. However, there is no one way to define or measure randomness in a hash function. These tests include the Strict Avalanche Criterion (SAC) Test, Cumulative Sums Test, Frequency (Monobit) Test, etc. With such a large number of diverse tests, all of which use distinct mathematical procedures, it is important to again note that a functions' passing of any one test does not definitively imply its randomness or security. As well, this paper will use the SAC Test to check for randomness, as it is the primary test which was used by NIST in their SHA-3 competition. We will now move on to defining the test, investigating the procedure behind it, and applying it to our scaled-down hash function.

Strict Avalanche Criterion (SAC) Test

We will begin by defining the avalanche effect:

Definition 3.2

The Avalanche Effect is the desirable property of cryptographic hash functions, wherein if an input is changed slightly, the output changes significantly.[10]

A function passes the Strict Avalanche Criterion Test if and only if whenever a single bit of the input is flipped (i.e.: a 0 bit is flipped to a 1 bit or vice versa), the resulting output has one-half of its bits being flipped/altered. To better understand how this test works, we will now consider a general case of the test. Note that there are multiple variations of this test, but for the purposes of this paper, we will be following the method described in the paper "Cryptographic Randomness Testing of Block Ciphers and Hash Functions" by Faith Sulak, et al.[11]

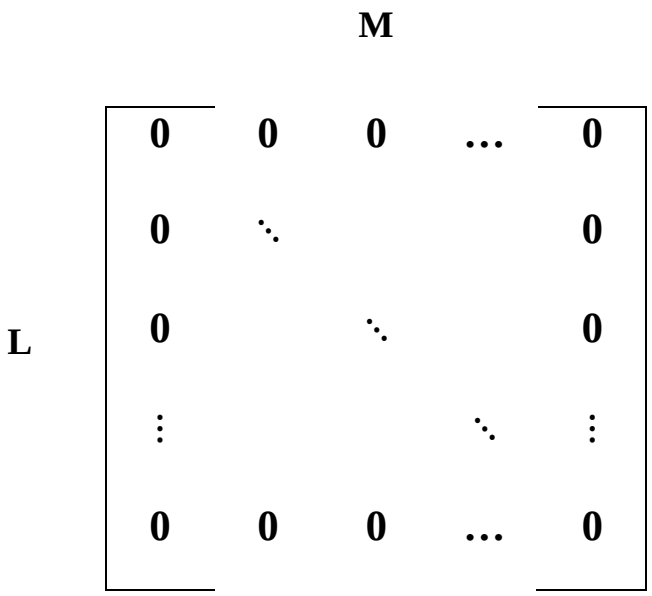
Consider the general case where we have:

- A sample size of **N** random strings
- Where each string has length of **L** bits
- A hash function with output key size of **M**

This test has 8-steps, which we will go through one-by-one.

Step 1)

Construct an L x M matrix and set every value of the matrix to zero:



Step 2)

Take the first string in our sample set and process it through the hash function.
Let X be our string. For example, let X = 010010…1 (Recall X has length L):

X —Hashing→ H(X) Ex: H(X)=101110…0 (Recall H(X) has length M)

Step 3)

Take our string X and flip the first bit, creating a new string X':

X' —Hashing→ H(X') Ex: H(X')=001010…1 (Recall H(X') has length M)

Step 4)

Next, we take our H(X) and H(X') values and XOR them:

XOR	H(X)	**EX: XOR**	101110…0
	H(X')		001010…1
	Y		100100…1

After XORing our two values, we arrive at a bitwise string, Y, where Y has length M. To the left we see an example of what this looks like.

Step 5)

Next, we take our resulting Y bitwise string and find every position, j, of the string which has a value of 1, where $1 \leq j \leq M$:

Ex: Y=100100…1

We can see above that each 1 bit in our Y-string has been underlined. Each one of these 1-values represent each positon in which our H(X) and H(X') differ.

Step 6)

Now, we add a '1' to every (i,j) entry in the matrix where we have a '1' in our string Y, where:

i is the bit we flipped in our original string X ($1 \leq i \leq L$)

j is the bit location of each 1 entry in the string Y ($1 \leq j \leq M$)

Ex: Y=100100…1

M

L

$$\begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{\dots} & \mathbf{1} \\ \mathbf{0} & \ddots & & & \mathbf{0} \\ \mathbf{0} & & \ddots & & \mathbf{0} \\ \vdots & & & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{\dots} & \mathbf{0} \end{bmatrix}$$

We can see above that for each ‘1’ value in our example Y string, we have added a ‘1’ to the first row of our matrix. We leave the rest of the values aside from the first row as ‘0’, since we have only flipped the first bit in our example string X.

Step 7)

If we continue this process, flipping each bit in our original string X and XORing it with our original string, we expect something similar to the following:

$$\mathbf{L} \quad \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \mathbf{M} \\ \begin{bmatrix} 1 & 0 & 0 & \dots & 1 \\ 1 & \ddots & & & 0 \\ 0 & & \ddots & & 1 \\ \vdots & & & \ddots & \vdots \\ 0 & 1 & 0 & \dots & 1 \end{bmatrix} \end{matrix}$$

We expect in the above matrix to have 50% ‘1’s and 50% ‘0’s, meaning each bit in our output hash flips 50% of the time.

Step 8)

If we again continue this process, running all N random strings in our sample set through the algorithm, we expect to arrive at the matrix:

M

L

N/2	**N/2**	**N/2**	**…**	**N/2**
N/2	⋱			**N/2**
N/2		⋱		**N/2**
⋮			⋱	⋮
N/2	**N/2**	**N/2**	**…**	**N/2**

Finally, we would expect our final matrix to have a value of N/2 in each entry. This again means that for our sample set of N random strings, whenever a single bit of the input is flipped, the resulting output has one-half (N/2) of its bits being flipped. To further understand this, let us consider and example.

Example 3.3

Consider the case where we have:

- A sample size of 100 random strings
- Each string having a length of 20 bits
- A hash function with an output key size of 80

Thus, for this example, we construct a 20x80 matrix and run the test for a sample size of 100 random strings, arriving at these results. Below, we can see two matrices. The first (Expected Outcome), depicts what our expected random matrix would look like for our given parameters. The second (Observed Outcome), shows some generic sample data we could get after running the test. In reality, for any hash function we run data through, we will not receive our exact expected outcome. In other words, no hash function is perfectly random, meaning that we will not get a value of N/2 in each matrix entry. Rather, we can see a varying range of numbers in each matrix position.

Expected Outcome

$$\begin{matrix} & 80 & & & & \\ 20 & \begin{bmatrix} 50 & 50 & 50 & \cdots & 50 \\ 50 & \ddots & & & 50 \\ 50 & & \ddots & & 50 \\ \vdots & & & \ddots & \vdots \\ 50 & 50 & 50 & \cdots & 50 \end{bmatrix} \end{matrix}$$

Observed Outcome

$$\begin{matrix} & 80 & & & & \\ 20 & \begin{bmatrix} 48 & 52 & 56 & \cdots & 41 \\ 49 & \ddots & & & 50 \\ 51 & & \ddots & & 49 \\ \vdots & & & \ddots & \vdots \\ 50 & 44 & 53 & \cdots & 57 \end{bmatrix} \end{matrix}$$

The question, now that we have obtained our observed data, is to analyze how random our data, and in turn our hash function is. Again, there are several ways to answer this question. For our purposes, we will be placing our data through the Chi-Squared Goodness of Fit Test, where the values in our observed outcome matrix should follow a binomial distribution.

We begin by first setting our Null Hypothesis to be that each matrix entry changes 50% of the time, meaning each outcome bit is flipped half of the time. In this way, we are setting our expected value to be N/2, or 50 in this specific example. Below is the formula for deriving our test statistic and degrees of freedom for our general case:

$$\text{Test Statistic} = \sum \frac{(\text{Observed} - \text{Expected})^2}{\text{Expected}} \qquad \text{Degrees of Freedom} = (L)(M)-1$$

For our test statistic, we go through each entry in our outcome matrix, plugging in each value into the formula and taking the overall sum. In terms of our degrees of freedom, we multiply the dimensions of our matrix, L and M, and subtract 1 from the total (L x M would be the total number of entries in our matrix). From this data, we obtain a P-value, $0 \leq P \leq 1$, wherein this value represents the evidence for or against our null hypothesis. In our case, the closer the P-value is to 1, the better our data reflects our hypothesis of each bit flipping one-half of the time. Our acceptance threshold (i.e.: our threshold for whether a function is random) is set a P-value of 0.01. Thus, if our resulting P-value ≥ 0.01, we have a random mapping. Otherwise, we conclude that our function is non-random.[11]

Applying this to our expected and observed data in Example 3.3, we obtain:

$$\text{Test Statistic} = \sum \frac{(48 - 50)^2}{50} + \frac{(52 - 50)^2}{50} + \ldots \qquad \text{Degrees of Freedom} = (20)(80)-1=1599$$

Say, from the above data, we end up with a P-Value = 0.13. Then we would conclude that we have a random mapping and that our hash function passes the Strict Avalanche Test. Now that we fully understand how this test works, we can begin to apply it to our own scaled-down hash function.

For our testing, we will have 4 different studies. In all four studies, we will have 64 rounds (as to keep the algorithms structure), and will have a string size of 160-bits which we will be inputting into the function. For studies 3 and 4, we are running the standard SHA-256 hash function through the SAC test in order to set a benchmark for how such a function would perform. As is the case for the standard SHA-256 function, in both cases we will have an outputting key size of 256-bits. In Study 1, we will be leaving the regular bitwise coefficients (of our bitwise rotates and shifts) unchanged. In the third study however, we will set the bitwise coefficients all to zero, meaning we will not perform any bitwise shifts or rotates. By doing this, we will hope to see the importance of the bitwise operations in our algorithm with regards to pre-image resistance. In studies 2 and 4, we are running our scaled-down hash function which we have developed. In doing this, we will see the differences in how the scaling down of an algorithm effects its performance in randomness testing. In both studies, we will have an outputting key size of 64-bits, rather than 256-bits, since we have scaled-down the function. In study 2, we will run our

scaled-down algorithm with its regular scaled-down bitwise coefficients. Recall that the values of these bitwise operations were determined through educated guesses. For example, if in the original SHA-256 function we performed a right-rotate of 16-bits, we would now in our 64-bit hash function perform a right-rotate of 4-bits, since a rotate of 16-bits on a 32-bit hash values is equivalent to a rotate of 4-bits on 8-bit hash values. In study 4, we will set our bitwise coefficients to zero (meaning no bitwise rotates or shifts). By doing this, we will again see this importance of the bitwise operations, even for a scaled-down function. Below is a table depicting this information:

SHA-256 Hash Function	Scaled-Down SHA-256 Hash Function
Study 1: ➢ Rounds: 64 ➢ Regular Bitwise Coefficients ➢ Key Size: 256-bits ➢ String Size: 20 (160-bits)	**Study 2:** ➢ Rounds: 64 ➢ Scaled-Down Bitwise Coefficients ➢ Key Size: 64-bits ➢ String Size: 20 (160-bits)
Study 3: ➢ Rounds: 64 ➢ Bitwise Coefficients Set To 0 ➢ Key Size: 256-bits ➢ String Size: 20 (160-bits)	**Study 4:** ➢ Rounds: 64 ➢ Bitwise Coefficients Set To 0 ➢ Key Size: 64-Bits ➢ String Size: 20 (160-bits)

Next, we will move on to the results of these studies.

Results of Strict Avalanche Criterion Test

Before we look at the results of our testing, it is important to note that these tests were each run for a sample size of 250 random sample strings, where each case we select from the following alpha-numeric set:

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

For each character selection in our sample string, we have 62 choices, which means that the total sample space is 62^{20}. The large size of the sample space ensures statistical accuracy.

A sample size of 250 random strings was chosen due to the limited computing power available (i.e.: Any larger sample size would result in unreasonable processing times). After running these tests, we arrive at the following results:

Results:

Study 1: (SHA-256 Regular Coefficients)	P-Value: 1 Conclusion: Random, Passes SAC Test
Study 2: (Down-Scaled SHA-256 Regular Coefficients)	P-Value: 1 Conclusion: Random, Passes SAC Test
Study 3: (SHA-256 Coefficients Set to Zero)	P-Value: 0 Conclusion: Non-Random, Fails SAC Test
Study 4: (Scaled-Down SHA-256 Coefficients Set to Zero)	P-Value: 0 Conclusion: Non-Random, Fails SAC Test

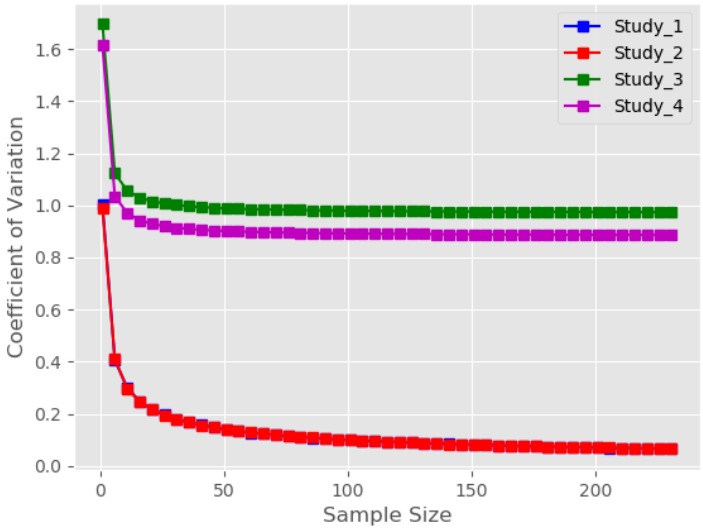
Thus, we arrive at a P-Value of 1 for studies 1 and 2, and a P-Value of 0 for studies 3 and 4. We conclude that studies 1 and 2 are random, whereas studies 3 and 4 are not random. From this, we see the importance of pre and post processing sections of our simplified hash function on the randomness of the output. These results match our initial hypothesis, since we know that bitwise operations in the pre and post processing, such as shifts and rotates, help to mix up the initial message. We further analyze these results by considering a number of different metrics.

Let us consider the Coefficient of Variation for our data. The Coefficient of Variation (CoV) is the ratio of the standard deviation to the mean. The higher the coefficient of variation, the greater the level of dispersion around the mean. The closer the value is to zero, the tighter the data is (i.e.: the smaller the standard deviation from the mean).

$$\text{Coefficient of Variation} = \frac{\text{Standard Deviation}}{\text{Mean}}$$

Below we see a plot of our results:

Plot 3.4

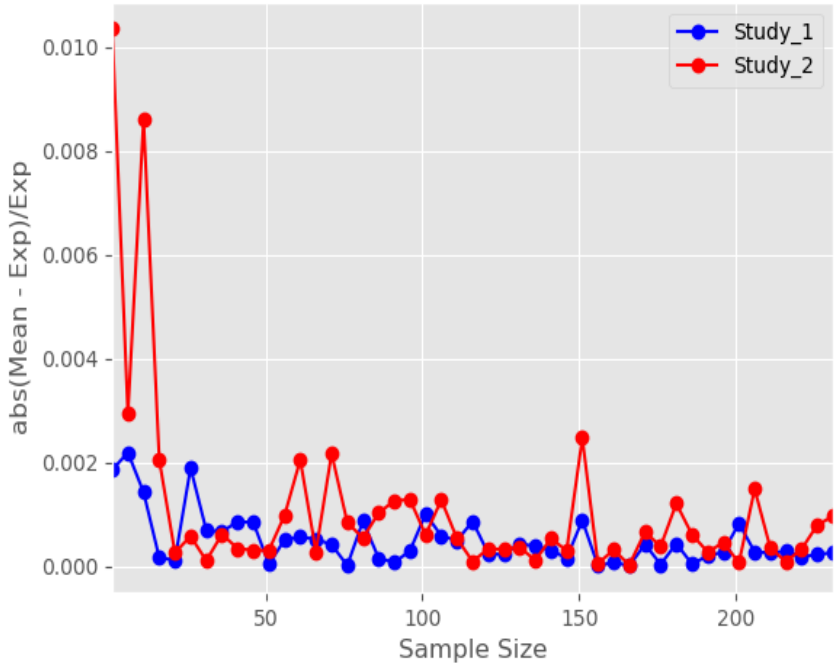


With regards to the above graph, we can clearly see the disparity between studies 1 and 2, where we apply regular bitwise coefficients, and studies 3 and 4, where these coefficients are set to zero. To overcome our limited computing power, we have calculated the Coefficient of Variation value for a varying number of sample sizes, as to see what value our CoV approaches. As our sample size increases, we expect the CoV to decrease, indicating a tightening around the mean. This is the case in studies 1 and 2. As the sample size increases, our CoV approaches zero, meaning that the deviation from the mean is minimal. This again confirms our original null hypothesis and shows further the randomness of the hash functions in studies 1 and 2. On the other hand, we can see that studies 3 and 4 have a much higher CoV, signalling that the deviation from the mean in our data is much higher. Again, the higher of CoV value is, the greater the level of dispersion around the mean, and the less random the function. Thus, from this we witness again the importance of the bitwise operations on the randomness of our hash functions.

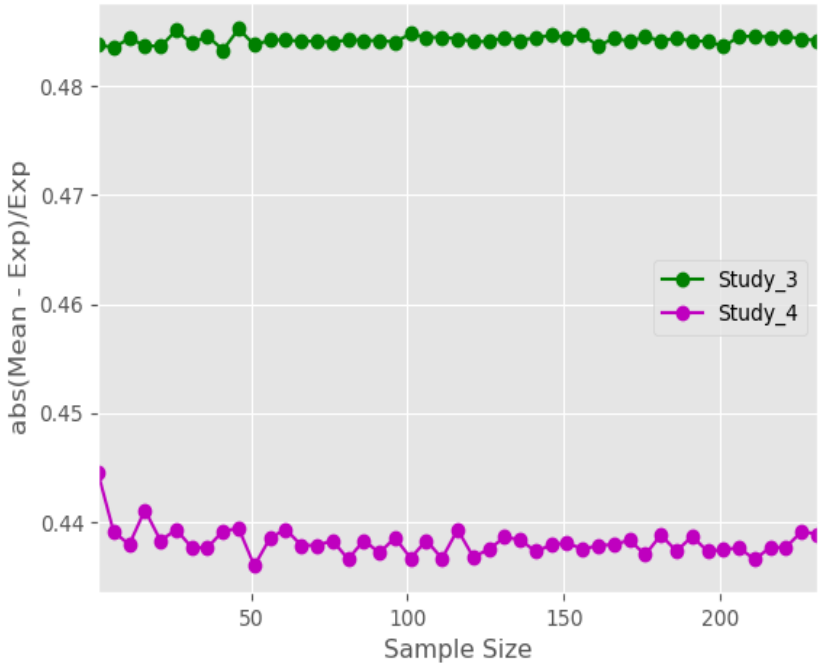
Another way to measure randomness is by comparing the expected mean to the observed mean of our entire matrix. Again, the closer our resulting value is to zero, the smaller the relative difference between our expected and observed mean. Thus, we consider a value closer to zero as being more random. Below is the equation and results of this test:

$$\frac{|\text{Observed Mean} - \text{Expected Mean}|}{\text{Expected Mean}}$$

Plot 3.5



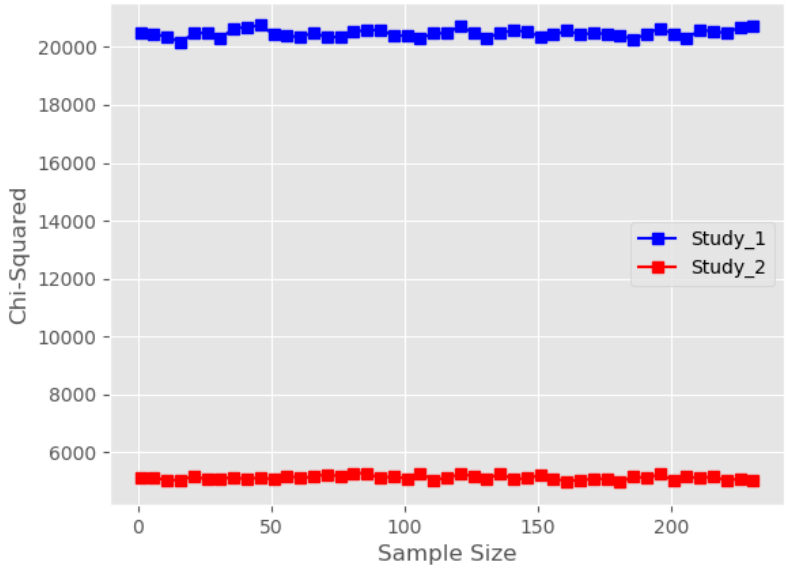
Plot 3.6



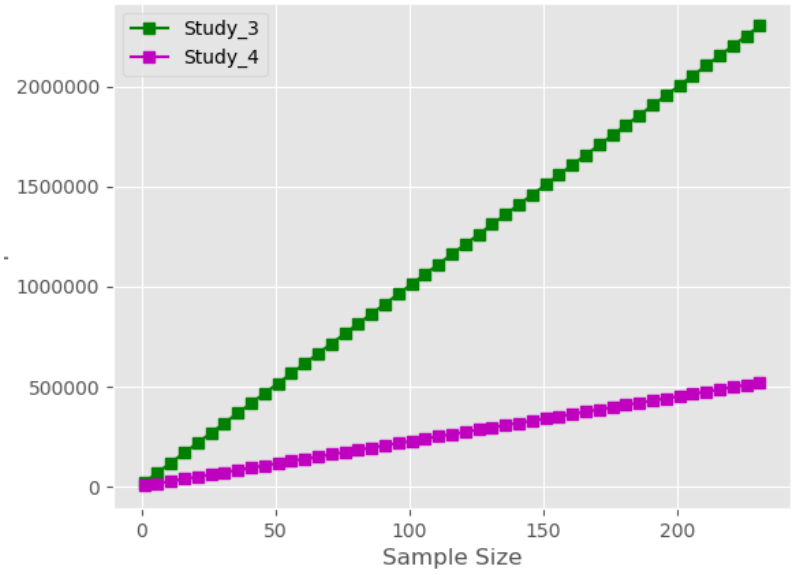
As we consider larger sample size, we expect the difference between the mean and the expected value to decrease. In this scenario, the expected value assumes that the hash function is completely random. As shown in Figure 3.5, the difference between the mean and the expected value falls to zero as the sample size is increased. This supports the hypothesis that the scaled down version of our function is random. However, this hypothesis does not hold for studies 3 and 4, where we have turned of pre/post processing. As shown in Figure 3.6, there is a non-zero value between the mean and the expected value, highlighting the fact that the hash function in studies 3 and 4 are not random.

Finally, we look to analyze the resulting Chi-Squared value of each function. We see the resulting plots below:

Plot 3.7



Plot 3.8



Plot 3.7 depicts study 1 and 2, which are our hash functions with regular pre/post processing. Plot 3.8 shows study 3 and 4, which are our hash functions with pre/post processing turned off. We see

that in Plot 3.7, as our sample size increases, our Chi-Squared values for both functions stays constant. Indicating randomness of the hash functions. However, in Plot 3.8, it is evident that as sample size increases, our hash functions increase linearly. This is a clear indication of non-randomness in the hash function from studies 3 and 4.

Conclusions Regarding Strict Avalanche Criterion Test

In the final analysis, through examination of our P-value, Coefficient of Variation, Difference in Means, and Chi-Squared values, we can confidently conclude that studies 1 and 2 have greater levels of randomness compared to studies 3 and 4. The outcome supports out hypothesis, as the use of bitwise operations in hash functions help to scramble the message, and in turn help to randomize the resulting hash value. We saw that even for our scaled-down function, the use of bitwise operations still helps to randomize our final hash. It is important to note, however, that in these studies we have taken the extremes (i.e.: No shifts or rotates compared to standard shifts and rotates). Since our standard bitwise coefficients produce a random function, and our coefficients set to zero produce a non-random function, we know that there is a breaking point at somewhere in between, where our scaled-down algorithm turns from non-random to random. We do this due to the high number of possible combinations for our coefficients, as it would take an unreasonable amount of time to complete all computations. It is worth mentioning as well that although certain studies may pass the SAC test, it does not necessarily imply that the function is secure or random. A functions' failing of any one of the hundreds of possible tests results in the function being classified as non-random and non-secure.

Next, we will look to test our scaled-down hash function for collision resistance.

Collision Resistance Testing

We will now consider the collision resistance applied to the scaled-down hash function. To begin, we will define what a collision is:

Definition 3.9
A collision takes place when two unique inputs, M_1 and M_2, for which $M_1 \neq M_2$, hash to the same value.

$$\text{A collision takes place if:} \quad H(M_1) = H(M_2) \text{ where } M_1 \neq M_2$$

The importance of collision resistance with regards to the Blockchain is clear to see when considering what hashing is used for. Let's consider an example:

Let T be the transaction "Don pays Roger 5 Bitcoins" and let T' be the transaction "Don pays Roger 5000 Bitcoins". Now consider the case where a block contains the transaction T. Say that this block hashes to a value X, and that Roger is able to create a separate block which replaces T with T'. If this block also hashes to X, then Roger would be able to place this new block into the Blockchain. By exploiting this collision, Roger would gain 4995 Bitcoins. Thus, we clearly see

the possible security hazards of a non-secure function. However, it is important to note that collisions are inevitable. Consider the fact that the number of unique possible inputs are practically limitless, whereas the number of unique hashing outputs is small and finite in comparison.

Consider the example of our scaled-down SHA-256 hash function. In our function, we have an output of 64-bits. The number of possibilities for our input string, as previously discussed, is practically limitless. Each input can range between 0 and 9 quadrillion bits, with the number of possible combinations of these bits being practically infinite. On the other hand, the possible combinations of outputs are only 2^{64}.

$$\text{Output (in bits): } \underline{2} \times \underline{2} \times \underline{2} \times \underline{2} \ldots = 2^{64}$$

Thus, we see the inevitability of collisions. However, our task, and the task of our hash function, is to limit these collisions. We look to limit these collisions by testing our functions for collision resistance. Next, we will discuss how these tests work.

Testing for Collisions:

To test for collisions, we follow a 6-Step process.

Step 1)

First, we initialize two empty arrays. Let us label S[ ] as the array which we will be storing our string values. Let F[ ] be the array in which we will be storing the corresponding hash values.

Step 2)

Next, we create a random string of length L, where, $1 \leq L \leq 100$. Let X be the random string that is created.

Step 3)

Once we have our random string, we process it through our hash function, obtaining a corresponding hash value. Let H(X) be the corresponding hash value.

Step 4)

Next, we append our X and H(X) values into the respective arrays S[ ] and F[ ]. Thus in our two arrays we will have the values S[X] and F[H(X)].

Step 5)

Now, we search through our hashing array, searching for two identical hashes. This is done once more than one random string has been hashed. We check the existing array for an identical hash value to the one we are appending. If no identical hash is found, we return to Step 2, where we

will create a new random string and continue the process. If an identical hash is found, we move to Step 6.

Step 6)

If two identical hashes are found, we compare our hash values to their corresponding string values in our array S. If the string values are unique, then we have a collision and we halt the process. If the string values are as well identical, then we do not have a collision, and return to Step 2.

Now that we have established how the test works, let us apply it to our 64-bit hash function.

Results for our scaled-down 64-bit hash function

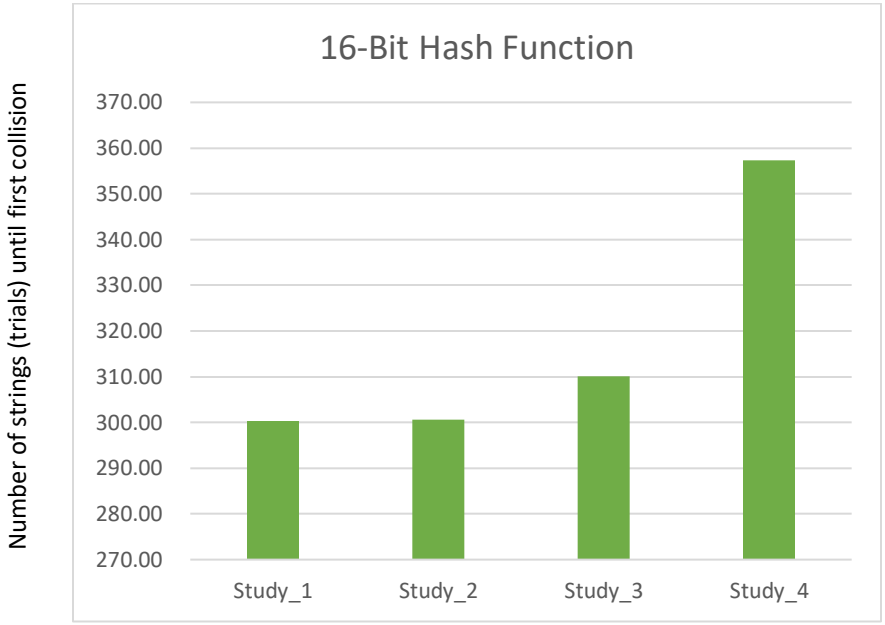
After running the collision test for 300,000 random strings (taking approximately one week in computing time) and multiple variations of bitwise shifts and rotates, no collisions were found. There are several possible explanations for this result. First, we must consider that, although scaled-down, we are nonetheless dealing with a standardized hashing algorithm (Recall that we have left the number of rounds in our schedule array and compression function unchanged). Since we have left the overall structure of the SHA-256 algorithm constant, it is fair to expect a strong performance from the algorithm in tests such as these. In addition, we must consider the fact that the output sample size is very large, of the order of 2^{64}. To expect a collision to take place in a small sample size of 300,000 random strings would be unreasonable. As well, it is important to note the significance of computing power in this test. A powerful computer, such as one in a technological institute, would be able to test a far greater set of random strings in a much shorter span of time, finding a collision much quicker than a personal computer. Thus, it is incorrect to make the assumption that our scaled-down hash function is collision resistant, due the lack of available computing power and time.

To overcome this roadblock in our testing, rather than altering the functions structure, we consider further scaling-down the algorithm. By doing so, we significantly reduce the size of our outputting set, making it more congruent to our respective computing power. We will consider both a 32-bit and 16-bit scaled-down hash function (the 32 and 16 bits referring to the outputting hash sizes). To scale-down to 32 and 16 bits, we follow the same procedure taken in the first section of this paper. For each function, we consider four studies, each of which is illustrated in the table below:

32-Bit Hash Function	16-Bit Hash Function
Study 1: • 64-Rounds • All Coefficients Set to Zero	Study 1: • 64-Rounds • All Coefficients Set to Zero
Study 2: • 64-Rounds • Right-Shifts = 2 Right-Rotates = 0	Study 2: • 64-Rounds • Right-Shifts = 1 Right-Rotates = 0
Study 3: • 64-Rounds • Right-Shifts = 0 Right-Rotates = 2	Study 3: • 64-Rounds • Right-Shifts = 0 Right-Rotates = 1
Study 4: • 64-Rounds • All Coefficients Set to 2	Study 4: • 64-Rounds • All Coefficients Set to 1

In each study, we run the collision test until a collision is found and confirmed. We continue doing this for a sample size of 100 tests and take the overall mean of our results. First, we consider the performance of our 16-bit hash function:

Plot 3.9

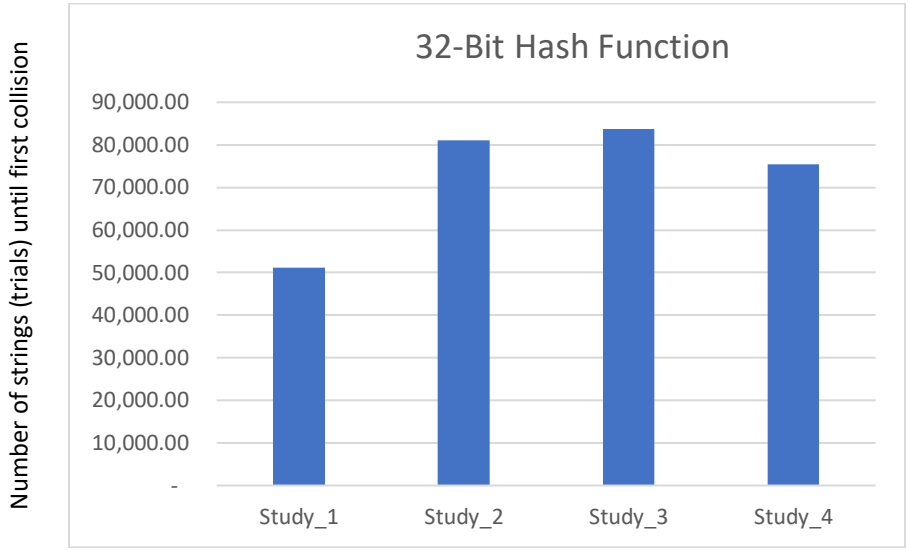


(Sample size of 100)

It is clear to see from these results the importance of bitwise shifting and rotating on a functions collision resistance. On average in Study 1, which performs zero rotations or shifts, a collision is found after only 300 random strings. However, as shifting and rotating is incorporated into the algorithm, the average number of trials until the first collision is found grows. Study 4, for

example, which has its bitwise coefficients set to 1, requires on average 357 random strings until the first collision is found. The above test is performed on the most scaled down version of the has function. The security of the hash function depends on the complexity and the size of the hashed chunks. As expected, it is not difficult to find collisions in such scenarios, as shown above.
Next, we consider our 32-bit function:

Plot 3.10



Once again, the significance of bitwise operations on a functions collision resistance is apparent. We see that as further bitwise operations are implemented on the function, the number of random strings until a first collision grows on average. Study 1, for example, takes only 51,000 trials on average, whereas the other three cases range from 76,000 to 84,000 average trials.
We note that the sample to establish a statistically sound result one must consider a lager sample space. Although our results can be used to define qualitative trends, they do not represent statistically accurate values.

Conclusion Regarding Collision Resistance Testing

In conclusion, both functions clearly demonstrate the importance of pre/post mixing operations on the collision resistance of our hash functions. It is clear that an extreme version of our scaled down hash function (16-bit) will not perform well under a collision test. This is also true for the 32-bit version of the hash function. There are two main properties of a hash function which protects it from brute force collision test. One is the complexity of the mixing algorithm. This is imbedded in the pre/post mixing processing section of the algorithm. The more we mix the input message, the harder it is to find a collision. Second is the size of the bit-chunks processed in the hash function. As we saw in our results, the larger the chunk size, the more difficult it was to find a collision. These two points highlight some of the important decisions when constructing new hash functions.

Section 4: Conclusion

In this thesis, we studied in detail the mechanics of the SHA-256 hash function which is at the heart of many security features of numerous technologies. We focused on the application of the SHA-256 hash function in Bitcoin, and Blockchain technology. As an alternative solution to current monetary systems, we explored case studies which highlighted the importance of secure ledger systems. The SHA-256 hash function is at the core of Blockchain technology. To explore its properties and test its security, we constructed a simplified/scaled down version of the hashing algorithm. In the scaled-down version of the hash function, we studied two critical features related to the algorithms security. The first, the functions Pre-Image resistance and its ability to produce a randomized outputting hash. The second, the functions Collision Resistance. In both cases, we discussed the real-life implementations and consequences of these properties with regards to be Blockchain. Through our implementation of the Strict Avalanche Criterion Test, we were able to see the importance of pre and post-processing with regards to a functions randomness. By implementing the Collision Test, we were able to better understand the significance of outputting key sizes, as well as pre and post-processing in a functions collision resistance. In the final analysis, through this work, we were able to create and implement an easier to understand scaled-down hash function. The use of statistical testing served to further our understanding a cryptographic hash functions' security, helping us develop more secure hash functions in the future.

Appendices

Appendix A

Character to Binary Conversion

A character is converted into binary (*0*'s and *1*'s) using the ASCII, which stands for the "American Standard Code of Information Interchange". Every character has a pre-determined decimal and binary value assigned to it[12]. Some examples include:

- "A" → *01000001*
- "!" → *00100001*
- "n" → *01101110*

Etc.…

Binary to Hexadecimal Conversion

Each byte, consisting of four bits, has an equivalent hexadecimal value, ranging from 0 to 9, and A to F, which is represented by the table below:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Appendix B

Bitwise Operations

Definition B.1

In the Right-Rotate operation, bits are "rotated" as if the left and right ends of the register were joined. The value that is shifted in on the left during a right-shift is whatever value was shifted out on the right[13].

Example B.1

Let us perform a Right-Rotate operation on the string *01010111* by 3 bits*:*

$$01010111 \longrightarrow 11101010$$

Thus, the resulting string becomes *11101010.*

Definition B.2

In an Arithmetic Right-Shift, the bits that are shifted out of the right end are discarded. Then, the sign bit is shifted in on the left, thus preserving the sign of the operand[13].

Example B.2

Let us perform an Arithmetic Right-Shift on the string *01010111* by 3 bits:

$$01010111 \longrightarrow 00001010$$

Thus, we can see that since the sign of our string in this case is a *0*, we shift-in three 0's from the left and discard the three farthest values on the right (in this case, *111*). The resulting string becomes *00001010*.

Remark B.1

It is important to note that information is lost when performing a bitwise Arithmetic Shift. In particular, the bits which are shifted out are lost, and it is near impossible to predict the lost information when trying to go backwards. For example, if 15-bits are shifted out in an Arithmetic Right-Shift, we know only the sign of the original string, but we do not know the value of any of the original shifted-out bits. Through this, we can observe one of several reasons why hash functions are one-way functions.

Definition B.3

A bitwise operation XOR takes two bit-patterns of equal length and performs the logical exclusive OR operation on each pairing[13].

A B	A XOR B
0 0	*0*
0 1	*1*
1 0	*1*
1 1	*0*

Example B.3

Let us perform an XOR operation on the two strings *01010101* and *11001100*.

01010101
11001100
10011001

Thus, we see that we get the resulting string of *10011001.*

Definition B.4

The bitwise NOT, or compliment, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Bits that are *0* become *1* and bits that are *1* become *0* [13].

A	NOT A
0	*1*
1	*0*

Example B.4

Let us perform a NOT operation on the string *11010011*.

11010011 ⟶ *00101100*

Thus, we see that we get the resulting string of *00101100.*

Definition B.5

The bitwise AND operation takes two equal-length binary representations and performs the logical AND on each pair of the corresponding bits by multiplying them. Thus, if both bits in the compared position are a *1*, the bit in the resulting binary representation is a *1*; otherwise, the result is a *0* [13].

A B	A AND B
0 0	*0*
0 1	*0*
1 0	*0*
1 1	*1*

Example B.5

Let us perform an AND operation on the strings *01010101* and *11001100*.

01010101
11001100
01000100

Thus, the resulting string becomes *01000100.*

Definition B.6

Bitwise Addition comprises of a binary number system which uses only two digits (*0* and *1*) and applies addition on two strings of equal length. There are four basic operations for binary addition [13].

A B	Addition (A+B)
0 0	*0*
0 1	*1*
1 0	*1*
1 1	*10*

Definition B.7

Binary Overflow takes place when the answer to an addition or subtraction problem in bits exceeds the magnitude which can be represented with the allowed number of bits[4].

Example B.6

Let us perform bitwise Addition on the strings *01010101* and *11001100* and take into consideration the Binary Overflow that takes place:

10010101
11001000
101011101

We can see that from the resulting bitwise Addition, we are left with a binary string of 9-bits. However, since we started by adding two 8-bit strings, we only have in our storage space room for an 8-bit string. This is again referred to as Binary Overflow and can be resolved by simply eliminating the extra bit at the front of our resulting string.

~~1~~01011101 ⟶ *01011101*

Thus, we are left with the resulting string of *01011101.*

References

1. Coindesk. “Bitcoin Price Index - Real-Time Bitcoin Price Charts.” CoinDesk, www.coindesk.com/price/.
2. “What is Bitcoin?” CNNMoney, Cable News Network, money.cnn.com/infographic/technology/what-is-bitcoin/.
3. Hamel, Angèle. “Bitcoin”, CP460 Applied Cryptography. Fall 2016 Class Notes.
4. Menezes, A. J., et al. Handbook of Applied Cryptography. CRC Press, 2001.
5. “What is a public Key?” SearchSecurity, searchsecurity.techtarget.com/definition/public-key.
6. “What is hashing?” SearchSQLServer, searchsqlserver.techtarget.com/definition/hashing.
7. Rogaway, Phillip, and Thomas Shrimpton. “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance.” Fast Software Encryption Lecture Notes in Computer Science, 2004. doi:10.1007/978-3-540-25937-4_24.
8. “9,223,372,036,854,775,807.” Wikipedia, Wikimedia Foundation, 22 Dec. 2017, en.wikipedia.org/wiki/9,223,372,036,854,775,807.
9. Regenscheid, A, et al. “Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition”. National Institute of Standards and Technology. September 2009.
10. Rogaway, P, Shrimpton, T. “Cryptographic Hash-Function Basics”. Fast Software Encryption. Vol. 3017. July 16, 2009.

11. Doganaksoy, A, Sulak, F. et al. “Cryptographic Randomness Testing of Block Ciphers and Hash Functions”, Institute of Applied Mathematics, Middle East Technical University.
12. “ASCII Table and Description.” Ascii Table – ASCII character codes and html, octal, hex and decimal chart conversion, www.asciitable.com/.
13. “Bitwise Operation.” Wikipedia, Wikimedia Foundation, 31 Dec. 2017, en.wikipedia.org/wiki/Bitwise_operation.