Statistical Analysis of the SHA-1 and SHA-2 Hash Functions

William Smith
Department of Computer Science
Rochester Institute of Technology
wjs3641@rit.edu

Abstract - The SHA-1 and SHA-2 hash functions are cryptographically secure hash functions designed by the National Security Agency (NSA) for use in many applications, such as message verification, security protocols and data integrity checks. One of the qualities a function must have to be cryptographically secure is an apparent randomness of the output when compared to the input message. This paper describes the analysis of the SHA-1 and SHA-2 hash function implemented with the CryptoStat library to determine the number of non-random rounds present in each function. The randomness (or lack thereof) of each round is determined by using various Bayesian model selection to calculate the odds ratio of a round’s randomness under a given statistical test.

I. INTRODUCTION

A hash function is a function designed to compress data of any length to an output of a fixed size called a message digest. Cryptographic hash functions are a special subset of hash functions which have been deemed secure enough for cryptography. To be deemed secure, the functions had to have five main properties. First, the function must be deterministic, in that every input message always returns the same message digest. Second, the computation of the message digest should be quick. Third, it should be infeasible to return to the input message from the message digest without a brute force attack against all possible messages. Fourth, the message digests of two similar but not equal message inputs should appear completely different. Fifth, it should be infeasible to find two input values to the hash function that generate the same message digests.

The Secure Hash Algorithm 1 (SHA-1) and Secure Hash Algorithm 2 (SHA-2) hash functions are cryptographic hash functions designed by the United States National Security Agency (NSA) used for various security protocols, data integrity and digital signatures. Since these hash functions should follow the principles listed earlier, the functions should generate message digests that are unique for each input, and appear to be random when compared to other digests. In order to confirm this apparent randomness however, it is necessary to use a suite of statistical tests comparing the message digests generated from each round of the algorithms.

The goal of this project is to analyze the SHA-1 and four of the six SHA-2 hash functions, and to determine the number of non-random rounds in each hash function. To do this, I use Professor Alan Kaminsky’s CryptoStat library, a Bayesian statistical test suite which utilizes NVIDIA’s CUDA parallel computing platform to evaluate the implemented hash functions. The project consisted of two main parts: the implementation of the hash functions, and the collection of data for each of the functions.

This paper is organized as follows. Section II reviews details on the CryptoStat library and the Bayesian statistical tests used for testing the randomness of the chosen hash functions. Section III defines the SHA-1 hash function, and its implementation, including the modifications made for this project. Section IV details the SHA-2 hash family, the four functions that are implemented for this project, and the modifications made to the functions for this project. Section V contains the set of tests run against each of the algorithms, and the conclusion based on the number of non-random rounds determined by the test suite. Finally, Section VI details potential future work related to this project.

II. BACKGROUND INFORMATION

a. CRYPTOSTAT LIBRARY

The CryptoStat library is a parallel Java and CUDA library used for the statistical analysis of cryptographic functions, such as block ciphers, hash functions, message authentication codes (MACs). The framework contains a number of Bayesian Odds Ratio tests which will be discussed in the second half of this section. The CryptoStat library contains two main programs, Analyze and AnalyzeSweep, that utilize a CUDA-capable GPU accelerated node to perform parallel computations.

Cryptographic functions written for the CryptoStat library must be designed such that they implement the Function superclass defined by the library. This superclass defines a set of methods, such as A and B input bit size methods and the evaluate method, which are required by the two main programs. The cryptographic functions must also implement a CUDA kernel which includes the FunctionKernel.cu file to operate on the GPU.

When run, the main program of the CryptoStat library takes four non-optional parameters: the cryptographic function to analyze, the A input Generator object, the B input Generator object, and the OddsRatio object used to analyze the function. In the case of the AnalyzeSweep program, the last three parameters are names of files that contain a list of the Generator objects or OddsRatio objects. The main program then initializes the CUDA kernel and sets up the appropriate memory locations to store the resulting output of the cryptographic function’s rounds. The output is stored in an array, with one entry for each round of the cryptographic function corresponding to the combination of the A and B input. Once the array has been populated with results from all possible combinations of A and B input Generator objects, the main program runs the given odds ratio test(s) again the results to determine the randomness of each round. The result is then displayed as the total number of non-random rounds from the cryptographic function. In the case of the AnalyzeSweep program, the maximum number of non-random rounds resulting from all possible test combinations is the displayed result.
b. STATISTICAL TESTS
The CryptoStat library implements the OddsRatio superclass as the abstract class for all Bayesian odds ratio tests that can be run against cryptographic functions. These OddsRatio tests are based on Bayesian model selection, which is detailed thoroughly in Kaminsky (2013) [XYZ]. For clarity however, these formulas are restated here in a condensed form.

Let H be a hypothesis which describes the expected outcome of a process and let D be a data sample taken from running said process. Using Bayes' Theorem, the conditional probability of H, given the data D is said to be

\[ pr(H | D) = \frac{pr(D | H)pr(H)}{pr(D)} \]  

(1)

where \( pr(D | H) \) is the probability of D, given that H has occurred and pr(H) and pr(D) are the probabilities of H and D.

The CryptoStat odds ratio tests takes two hypotheses H₁ and H₂ where \( pr(H_1) \) is the probability that the population is random, and \( pr(H_2) \) is the probability that the population is non-random. After collecting the data D, the posterior odds ratio of the hypotheses H₁ and H₂ can be calculated from Equation (1) as

\[ \frac{pr(H_1|D)}{pr(H_2|D)} = \frac{pr(D|H_1)pr(H_1)}{pr(D|H_2)pr(H_2)} \]  

(2)

where \( \frac{pr(H_1)}{pr(H_2)} \) is the odds ratio of the previous models, and \( \frac{pr(D|H_1)}{pr(D|H_2)} \) is the Bayes Factor.

While the CryptoStat library is based on the odds ratio tests, the formulas used in CryptoStat contain parameters \( \theta \), and therefore the conditional probabilities are generated by calculating the integral of the possible parameter values, as shown in Kaminsky [XYZ]. Using these parameterized probabilities, it is possible to calculate the odd ratio using the following equation

\[ \frac{pr(D|H_1)}{pr(D|H_2)} = \frac{\Gamma(n+2)}{\Gamma(k+1)^k(n-k)^{n-k}} p^k (1-p)^{n-k} \]  

(3)

as derived in Kaminsky [XYZ]. However, as Kaminsky states, the gamma function has the change of overflowing the range of values, so the CryptoStat library computes the logarithm of the Bayes Factor. As a result, the odds ratio probabilities generated by CryptoStat can be easily observed as passing or failing the randomness test by observing the resulting ratio. If the ratio is greater than 0, the test is passed, and the hypothesis H₁ holds true. If the resulting ratio is less than 0, the test is failed. In the case of odds ratio results for this project, passing the test means that the cryptographic function shows random behavior for the given test if the ratio is greater than 0, or shows non-random behavior if the ratio is less than 0.

III. SECURE HASH ALGORITHM 1 (SHA-1)
SHA-1 is a cryptographic hash function developed by the NSA and published in 1995 as part of Federal Information Processing Standards (FIPS) Publication 180-1 entitled “Secure Hash Standard.” The algorithm was originally designed for use as part of the Digital Signature Algorithm (DSA) to generate or verify a signature for a given message, but it’s applications can be expanded to other security protocols or for use as a verification of data integrity.

The SHA-1 function is broken up into three main steps. The first phase of the function, message padding, takes place to force the message input to have a total length that is a multiple of 512 bits. Once the message has a total length that is a multiple of 512, the message is broken up into 512 bit blocks, and these blocks are processed by the message schedule expansion phase and the compression function phases in sequence. In a full implementation of SHA-1, the intermediate digest of each message block is added to the previous until the full message has been processed. For this paper however, a modified version of SHA-1 has been implemented, which will be discussed later in this section.

a. INITIAL VALUES AND ROUND CONSTANTS
The SHA-1 function uses the following five 32-bit words denoted as \( H_0 \) through \( H_4 \) as initial values for the message digest.

\[ H_0 = 0x67452301 \]
\[ H_1 = 0xEFCDAB89 \]
\[ H_2 = 0x98BADCFE \]
\[ H_3 = 0x10325476 \]
\[ H_4 = 0xC3D2E1F0 \]

In addition to the five \( H \) values, SHA-1 uses one of four round constants in each round of the compression function. The round constants at 32-bit values, and the round constant used is based on the current round.

\[ K_0 = 0x5A827999 \] (\( 0 \leq \text{round} \leq 19 \))
\[ K_1 = 0x6ED9BAE1 \] (\( 20 \leq \text{round} \leq 39 \))
\[ K_2 = 0x8F119784 \] (\( 40 \leq \text{round} \leq 59 \))
\[ K_3 = 0xCA62C1D6 \] (\( 60 \leq \text{round} \leq 79 \))

b. MESSAGEPadding
Messages evaluated by SHA-1 are considered to be a bit strings consisting entirely 1s or 0s, where the length of the message is the number of bits in the bit string. To create message blocks of exactly size 512 bits, the SHA-1 function must first pad the input message. The process of message padding for SHA-1 is a three-step process which can be applied to a message of any length to prepare it to be hashed by SHA-1.

First, the original message has a ‘1’ bit appended to the message. Next, \( k ‘0’ \) bits are appended to the message, such that the resulting message length \( l \) can be written as

\[ l = 448 \mod 512 \]

Finally, the length of the original message, denoted as \( l \), is calculated as a 64-bit big-endian integer. Once calculated, this value is appended to the new message, resulting in a padded message with a total length that is a multiple of 512.

c. MESSAGE SCHEDULE EXPANSION
Once the message has been padded properly, the function moves on to the message schedule phase. First, the padded message is broken into \( n \) 512-bit blocks. For each block, an array, denoted as \( w \) henceforth, of eighty 32-bit words is initialized and processed by the compression function. The message block is split into sixteen 32-bit words and stored in the first sixteen indices of \( w \). To extend the first sixteen words
section III

d. MESSAGE DIGEST COMPUTATION

Once the message schedule for a given block has been expanded, the compression function generates an intermediate digest for the block. First, the compression function initializes five 32-bit words denoted as $A$ through $E$. These values are set to the current value of the message digest, stored in $H_0$ through $H_6$. Once the $A$ through $E$ have been initialized, the function executes the algorithm shown in Figure 2 for all 80 rounds of the function.

![Figure 2: Single Round of SHA-1 Compression](image)

In Figure 2, the value of $A$ for each round is decided by the sum of $F(B, C, D)$, $\text{LeftRotate}(A, 5)$, $w_t$, and $k_t$ modulo 32, where $F$ is a non-linear function defined for each round. The values of the other four words are moved down the line without modification, with the expectation of $B$ which is rotated left by 30 bits.

Once all 80 rounds of compression have been executed on the message block, the values of $A$, $B$, $C$, $D$, and $E$ are added to $H_0$, $H_1$, $H_2$, $H_3$, and $H_4$ respectively. Once the entire message has been compressed, the values $H_0$ through $H_4$ are concatenated and returned as the 160-bit message digest.

e. CRYPTOStat IMPLEMENTATION MODIFICATION

For consistency across all test cases, my implementation of SHA-1 has been modified from the full-fledged implementation. The details of the initialization values, compression function and non-linear bitwise operation functions have remained the same, but the input to the function is limited to 416 bits.

The CryptoStat library defines an $A$ and $B$ input, and allows the implemented function to define the bit size of each input. For this project, $A$ is defined as 256 bits and $B$ is defined as 160 bits. Once the input is collected, the message padding phase of SHA-1 appends three hard-coded 32-bit words, following the message padding phase details, to create the full 512-bit message block.

In addition to the modification to restrict the input to the SHA-1 function, my implementation of SHA-1 also calculates the intermediate hash of the message block for each round. Once the value is calculated, it is stored in the output array $C$ defined by CryptoStat, because the hash value for each round is required for the randomness tests done by the Odds Ratio objects.

IV. SECURE HASH ALGORITHM 2 (SHA-2)

SHA-2 is a family of cryptographic hash functions that were designed by the NSA to compliment the SHA-1 hash function. The family of functions were first published in 2001, and the publication contained details for the six versions of SHA-2 contained in the set. This project only focused on the primary four SHA-2 algorithms (SHA-224, SHA-256, SHA-384, and SHA-512), and did not deal with the truncated functions (SHA-512/224 and SHA-512/256). The differences between the four variants implemented in this project will be described in Section IV subsection d, and for the rest of this section, SHA-256 will be the focus.

SHA-256 stands as the basis for the SHA-2 set of hash functions, relies on three main phases, similar to SHA-1. The three main phases for the SHA-256 algorithm are the message padding phase, the message schedule expansion phase, and finally the compression functions rounds. While the phases of SHA-2 seem identical to SHA-1, SHA-2 includes significant changes from the SHA-1 function, and resulting digests from any of the family’s functions are completely distinct from SHA-1 message digests.

a. INITIAL VALUES AND ROUND CONSTANTS

SHA-256 uses sixty-four 32-bit words as round constants, where each word is used for a single round of the compression function. The words are calculated by taking the first 32 bits of the fractional parts of the cube roots of the first sixty-four prime numbers, as documented in FIPS 180-4 [FIPS XYZ]. The initial hash values for SHA-256 are similarly obtained, as they are taken from the first 32 bits of the fractional parts of the square roots of the first eight prime numbers [FIPS XYZ]. Both the round constants and initial hash values are documented below in Figure 3.

![Figure 3: SHA-256 Round Constants and Initial Hash Values](image)

b. MESSAGEpadding

The message padding for SHA-256 is identical to the message padding for SHA-1, as documented in Section III subsection b.

c. MESSAGE SCHEDULE EXPANSION

Once the message has been padded properly, the function moves on to the message schedule phase. First, the padded message is broken into $n$ 512-bit blocks. For each block, an
array, denoted as \( w \) henceforth, of sixty-four 32-bit words is initialized and processed by the compression function. The message block is split into sixteen 32-bit words and stored in the first sixteen indices of \( w \). To extend the first sixteen words into the full sixty-four words, the message schedule then follows the algorithm in Figure 4.

```c
for (int i = 16; i < 64; i++)
    int s0 = rightRotate(v[i-16], 7) ^ rightRotate(v[i-16], 18) ^ v[i-16] >> 3);
    int s1 = rightRotate(v[i-9], 17) ^ rightRotate(v[i-16], 19) ^ v[i-16] >> 10;
    w[i] = w[i-16] + w[i-7] + x1;
```

*Figure 4: SHA-256 Message Schedule Expansion*

d. **MESSAGE DIGEST COMPUTATION**

After the message schedule has been expanded, the compression function is run for 64 rounds to compute the message block’s digest. The compression function initializes eight working variables, each a 32-bit word, and sets them to the current values of \( H_0 \) through \( H_7 \). Once the values, denoted as \( A \) through \( H \), are initialized, the compression runs the algorithm depicted in Figure 5 for sixty-four rounds.

*Figure 5: Single Round of SHA-256 Compression Function*

In Figure 5, the \( Ch \), \( Maj \), \( \Sigma_t \) and \( \Sigma_o \) functions are bitwise operations performed on the inputs given. Once the functions have calculated the results, those function outputs are added to \( H \), \( w_t \) and \( k \) modulo 32 to calculate the next \( A \) value.

After the 64 rounds of SHA-256 have finished, the block’s message digest is calculated by adding \( A \) through \( H \) with \( H_0 \) through \( H_7 \). After all message blocks have been processed, the message digest for the entire message is the concatenated result of \( H_0 \) through \( H_7 \).

e. **SHA-2 VARIANTS**

As mentioned previously, SHA-2 is a set of six hash functions, with each function having slightly different properties. The basic properties of each of the functions can be seen in Table 1, with algorithm specific details following.

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Block Size (bits)</th>
<th>Word Size (bits)</th>
<th>Digest Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-224</td>
<td>512</td>
<td>32</td>
<td>224</td>
</tr>
<tr>
<td>SHA-256</td>
<td>512</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>SHA-384</td>
<td>1024</td>
<td>64</td>
<td>384</td>
</tr>
<tr>
<td>SHA-512</td>
<td>1024</td>
<td>64</td>
<td>512</td>
</tr>
</tbody>
</table>

*Table 1: SHA-2 Function Properties*

As seen in Table 1, the message digest size directly corresponds to the name of the SHA-2 hash function. In addition, the table illustrates that while SHA-224 and SHA-256 operate on 32-bit words (such as the constants and the message schedule), all of the operations for SHA-384 and SHA-512 operate on 64-bit words. Since the word size doubles for SHA-384 and SHA-512, the message block size is also doubled.

SHA-224 and SHA-256 differ in two very small ways. First, SHA-224 omits \( H_r \) from the end digest to get a digest size of 224 bits. Second, SHA-224 uses different initialization values for the starting hash value. SHA-224 opts to use the ninth through sixteenth prime number to generate the fractional part of the square root for initial values.

In addition to the two properties listed above, SHA-512 and SHA-256 differ in several more key characteristics. First, SHA-512 uses 80 rounds of compression as opposed to SHA-256’s 64 rounds. As such, the message schedule for SHA-512 contains 80 words instead of 64 words. Next, the message padding for SHA-512 uses a 128-bit representation of the original message length \( l \), and the congruence equation is \( l \equiv 896 \mod 1024 \). The round constants and initial hash values used in SHA-512 are 64-bit words, and therefore different values from SHA-256. Finally, the shift and rotate values that are used for message schedule expansion and during the compression function rounds are different for SHA-512.

The SHA-384 function is very similar to the SHA-512 function, with only two modifications. First, the initial hash values are derived from the ninth through sixteenth prime numbers, like SHA-224. Second, the values \( H_5 \) and \( H_7 \) are omitted from the end digest in order to return a digest of length 384.

f. **CRYPTOStat IMPLEMENTATION MODIFICATIONS**

Similar to SHA-1, the implementations of SHA-224, SHA-256, SHA-384 and SHA-512 used for this project are restricted and modified version of the full function. For SHA-224 and SHA-256, the bit size of input A is restricted to 256 bits, and the bit size of input B is 160 bits. For SHA-384 and SHA-512, the bit size of input A is restricted to 512 bits, and the bit size of input B is 320 bits.

In addition, each of these SHA-2 hash functions calculate the intermediate message digest to store results from each round of the compression function. These computations do not affect the end message digest however, and resulting digsests from my implementations are identical to the output from the official implementations.

V. RESULTS

The results from testing the implementations of SHA-1 and SHA-2 for this paper will be broken into two parts. First, the detailed results of a single test case against SHA-256 are shown and explained in subsection a. In subsection b, the full test suite that was used to conduct extensive testing against each of the functions is documented with the result of the maximum number of non-random rounds for each function.

a. **SINGLE SHA-256 TEST AND RESULTS**

The small test case run to show exactly what happens during each test of the larger test suite was run using the Analyze main program of the CryptoStat library. The Analyze main program takes in the name of the function to run, a Generator object for input A, a Generator object from input B,
and an Odds Ratio object. For this test, I used the A input Generator “Gray(256),” the B input Generator “Shift(8,Gray(256))” and the Odds Ratio object “Difference(2).”

The Generator object “Gray(N)” is a Generator object that creates N different A input values, starting at the initial value 0, and incrementing by one bit until N values have been generated. The “Shift(N,G)” Generator takes a Generator G, and shifts the resulting output to the left by N bits. Finally, the Odds Ratio object “Difference(G)” computes the odds ratio for the hypothesis that the bitwise exclusive-or of a bit group with size G in one output and that bit group in the previous output is uniformly distributed [Kaminsky XYZ].

Table 2 displays the aggregate result from all A and all B inputs from the generators described above. Please note that to preserve space, rounds 33 through 63 are omitted as the results continue the pattern of positive odds ratio values. The full command used to run the test is as follows:

```
java pj2 Analyze -nt=768 "SHA256()" "Gray(256)"
"Shift(8,Gray(256))" "Difference(2)"
```

<table>
<thead>
<tr>
<th>Round</th>
<th>Odds Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>2</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>3</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>4</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>5</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>6</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>7</td>
<td>-1.1627 * 10^7</td>
</tr>
<tr>
<td>8</td>
<td>-1.1621 * 10^7</td>
</tr>
<tr>
<td>9</td>
<td>-1.1595 * 10^7</td>
</tr>
<tr>
<td>10</td>
<td>-1.1544 * 10^7</td>
</tr>
<tr>
<td>11</td>
<td>-1.1496 * 10^7</td>
</tr>
<tr>
<td>12</td>
<td>-1.1453 * 10^7</td>
</tr>
<tr>
<td>13</td>
<td>-1.0969 * 10^7</td>
</tr>
<tr>
<td>14</td>
<td>-8.7681 * 10^6</td>
</tr>
<tr>
<td>15</td>
<td>-5.9545 * 10^6</td>
</tr>
<tr>
<td>16</td>
<td>-3.0894 * 10^6</td>
</tr>
<tr>
<td>17</td>
<td>-7.0386 * 10^5</td>
</tr>
<tr>
<td>18</td>
<td>-51208</td>
</tr>
<tr>
<td>19</td>
<td>-66.361</td>
</tr>
<tr>
<td>20</td>
<td>578.75</td>
</tr>
<tr>
<td>21</td>
<td>581.16</td>
</tr>
<tr>
<td>22</td>
<td>560.90</td>
</tr>
<tr>
<td>23</td>
<td>585.61</td>
</tr>
<tr>
<td>24</td>
<td>560.07</td>
</tr>
<tr>
<td>25</td>
<td>549.49</td>
</tr>
<tr>
<td>26</td>
<td>555.08</td>
</tr>
<tr>
<td>27</td>
<td>582.64</td>
</tr>
<tr>
<td>28</td>
<td>573.65</td>
</tr>
<tr>
<td>29</td>
<td>567.76</td>
</tr>
<tr>
<td>30</td>
<td>585.95</td>
</tr>
<tr>
<td>31</td>
<td>573.98</td>
</tr>
<tr>
<td>32</td>
<td>584.13</td>
</tr>
<tr>
<td>64</td>
<td>570.66</td>
</tr>
</tbody>
</table>

**Table 2: SHA-256 Difference Test Results**

From the results in Table 2, and remembering back to the details on Bayesian Odds Ratio calculations from Section II, one can tell that for this Difference Test using bit groups of two, there are 19 non-random rounds, and the rest of the rounds are random. This is because the odds ratio values for rounds 1 through 19 are all negative values, showing that across all A and B inputs, the difference test found behavior that did not match the hypothesis of uniformly distributed bits of the output for that round and the previous round. Starting on round 20 however, the resulting odds ratio is positive, meaning that starting with this round, the hypothesis is likely to be true, and the round determined to be random.

The Analyze main program can quickly determine the number of non-random rounds for a single test case, but the number of non-random rounds for a given function could be low for one test case, and high for a different case. To determine the maximum number of non-random rounds in the function, it is best to test the functions under a large number of test cases to rule out any outliers. This process is documented in the next subsection.

b. FULL TEST SUITE AND RESULT

To fully test each of the five hash functions implemented for this project, I used the CryptoStat main program AnalyzeSweep. As mentioned earlier in section II, the AnalyzeSweep program takes a file containing a list of A Generators, B Generators and Odds Ratio objects. For my testing, I used a list of 36 A input Generators, 100 B input Generators, and 39 Odds Ratio objects. For a fully listing of the Generators and Odds Ratio objects, please see the Appendix in Section VIII.

Using this test suite, I ran AnalyzeSweep with the describe files against SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 to collect the results shown in Table 3. Each test took approximately 60 hours running on an NVIDIA Tesla K40c GPU accelerator using all 2880 cores available. One important note is that the tests for SHA-384 and SHA-512 took approximately four or five times longer due, and due to time constraints, the non-random round results shown for these functions only takes the first 9 A input Generators into consideration. This is denoted by an asterisk next to the affected test results.

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>Number of Non-Random Rounds</th>
<th>Total Number of Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>26</td>
<td>80</td>
</tr>
<tr>
<td>SHA-224</td>
<td>18</td>
<td>64</td>
</tr>
<tr>
<td>SHA-256</td>
<td>19</td>
<td>64</td>
</tr>
<tr>
<td>SHA-384</td>
<td>18*</td>
<td>80</td>
</tr>
<tr>
<td>SHA-512</td>
<td>20*</td>
<td>80</td>
</tr>
</tbody>
</table>

**Table 3: AnalyzeSweep Full Test Results**

From these results, it should be noted that all five of the hash functions fall between 22% and 33% of their rounds being non-random, with the best ratio being SHA-384 with 22.5% non-random rounds, and the worst being SHA-1 with 32.5% non-random rounds. It is important to note however, that the number of non-random rounds for a given algorithm does not imply anything about the strength of the algorithm, only gives information to a weakness. If the algorithm shows more random rounds than non-random rounds to a certain extent, which all five SHA functions implemented here abide by, the strength of the output cannot be determined by these statistical tests, and further cryptanalysis would be needed.

In conclusion, based on the extensive test cases run against the SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512...
implementations for this project, each of the five algorithms have shown an appropriate number of random rounds compared to their non-random rounds.

VI. FUTURE WORK

Future work in line with the experiment described in this paper could include implementation of the truncated SHA-2 functions, NIST’s SHA-3 hash function and SHAKE pseudo-random number generator (PRNG) or implementations of varying other cryptographic functions. In addition, further research into odds ratio tests not implemented by the CryptoStat library is another avenue of future work. Finally, the experiment results of number of non-random rounds in the SHA-1 and SHA-2 algorithms could be expanded to research the implications of a cryptographic function’s design on the number of rounds necessary to achieve a desirable amount of apparent randomness in the resulting output.

VII. REFERENCES


VIII. APPENDIX

Contained below are the listing of Generators and Odds Ratio objects used for the Analyze Sweep program. Each set of Generators or Odds Ratio objects were contained in a file, and the three files were provided to the AnalyzeSweep program as parameters. The generator and object lists were provided by Professor Alan Kaminsky, and were designed by him while completing tests for his own research. They have been used and modified with his permission.

A INPUT GENERATOR LIST:

Gray(256)
Shift(8,Gray(256))
Shift(16,Gray(256))
Shift(24,Gray(256))
Shift(32,Gray(256))
Shift(40,Gray(256))
Shift(48,Gray(256))
Shift(56,Gray(256))
Shift(64,Gray(256))
Shift(72,Gray(256))
Shift(80,Gray(256))
Shift(88,Gray(256))
Shift(96,Gray(256))
Shift(104,Gray(256))
Shift(112,Gray(256))
Shift(120,Gray(256))
Complement(Gray(256))
Complement(Shift(8,Gray(256)))
Complement(Shift(16,Gray(256)))
Complement(Shift(24,Gray(256)))
Complement(Shift(32,Gray(256)))
Complement(Shift(40,Gray(256)))
Complement(Shift(48,Gray(256)))
Complement(Shift(56,Gray(256)))
Complement(Shift(64,Gray(256)))
Complement(Shift(72,Gray(256)))
Complement(Shift(80,Gray(256)))
Complement(Shift(88,Gray(256)))
Complement(Shift(96,Gray(256)))
Complement(Shift(104,Gray(256)))
Complement(Shift(112,Gray(256)))
Complement(Shift(120,Gray(256)))
OneOff()
Xor(55555555555555555555555555555555,OneOff())
Xor(AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA,OneOff())
Xor(FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF,OneOff())
B INPUT GENERATOR LIST:

Gray(256)
Shift(8,Gray(256))
Shift(16,Gray(256))
Shift(24,Gray(256))
Shift(32,Gray(256))
Shift(40,Gray(256))
Shift(48,Gray(256))
Shift(56,Gray(256))
Shift(64,Gray(256))
Shift(72,Gray(256))
Shift(80,Gray(256))
Shift(88,Gray(256))
Shift(96,Gray(256))
Shift(104,Gray(256))
Shift(112,Gray(256))
Shift(120,Gray(256))
Shift(128,Gray(256))
Shift(136,Gray(256))
Shift(144,Gray(256))
Shift(152,Gray(256))
Shift(160,Gray(256))
Shift(168,Gray(256))
Shift(176,Gray(256))
Shift(184,Gray(256))
Shift(192,Gray(256))
Shift(200,Gray(256))
Shift(208,Gray(256))
Shift(216,Gray(256))
Shift(224,Gray(256))
Shift(232,Gray(256))
Shift(240,Gray(256))
Shift(248,Gray(256))
Shift(256,Gray(256))
Shift(264,Gray(256))
Shift(272,Gray(256))
Shift(280,Gray(256))
Shift(288,Gray(256))
Shift(296,Gray(256))
Shift(304,Gray(256))
Shift(312,Gray(256))
Shift(320,Gray(256))
Shift(328,Gray(256))
Shift(336,Gray(256))
Shift(344,Gray(256))
Shift(352,Gray(256))
Shift(360,Gray(256))
Shift(368,Gray(256))
Shift(376,Gray(256))
Complement(Gray(256))
Complement(Shift(8,Gray(256)))
Complement(Shift(16,Gray(256)))
Complement(Shift(24,Gray(256)))
Complement(Shift(32,Gray(256)))
Complement(Shift(40,Gray(256)))
Complement(Shift(48,Gray(256)))
Complement(Shift(56,Gray(256)))
Complement(Shift(64,Gray(256)))
Complement(Shift(72,Gray(256)))
Complement(Shift(80,Gray(256)))
Complement(Shift(88,Gray(256)))
Complement(Shift(96,Gray(256)))
Complement(Shift(104,Gray(256)))
Complement(Shift(112,Gray(256)))
Complement(Shift(120,Gray(256)))
Complement(Shift(128,Gray(256)))
Complement(Shift(136,Gray(256)))
Complement(Shift(144,Gray(256)))
Complement(Shift(152,Gray(256)))
Complement(Shift(160,Gray(256)))
Complement(Shift(168,Gray(256)))
Complement(Shift(176,Gray(256)))
Complement(Shift(184,Gray(256)))
Complement(Shift(192,Gray(256)))
Complement(Shift(200,Gray(256)))
Complement(Shift(208,Gray(256)))
Complement(Shift(216,Gray(256)))
Complement(Shift(224,Gray(256)))
Complement(Shift(232,Gray(256)))
Complement(Shift(240,Gray(256)))
Complement(Shift(248,Gray(256)))
Complement(Shift(256,Gray(256)))
Complement(Shift(264,Gray(256)))
Complement(Shift(272,Gray(256)))
Complement(Shift(280,Gray(256)))
Complement(Shift(288,Gray(256)))
Complement(Shift(296,Gray(256)))
Complement(Shift(304,Gray(256)))
Complement(Shift(312,Gray(256)))
Complement(Shift(320,Gray(256)))
Complement(Shift(328,Gray(256)))
Complement(Shift(336,Gray(256)))
Complement(Shift(344,Gray(256)))
Complement(Shift(352,Gray(256)))
Complement(Shift(360,Gray(256)))
Complement(Shift(368,Gray(256)))
Complement(Shift(376,Gray(256)))
OneOff()
ODDS RATIO OBJECTS:

Uniformity(1)
Uniformity(2)
Uniformity(2,193562225603L)
Uniformity(2,641548912186L)
Uniformity(2,360639399813L)
Uniformity(4)
Uniformity(4,193562225603L)
Uniformity(4,641548912186L)
Uniformity(4,360639399813L)
Uniformity(8)
Uniformity(8,193562225603L)
Uniformity(8,641548912186L)
Uniformity(8,360639399813L)
Avalanche(1)
Avalanche(2)
Avalanche(2,193562225603L)
Avalanche(2,641548912186L)
Avalanche(2,360639399813L)
Avalanche(4)
Avalanche(4,193562225603L)
Avalanche(4,641548912186L)
Avalanche(4,360639399813L)
Avalanche(8)
Avalanche(8,193562225603L)
Avalanche(8,641548912186L)
Avalanche(8,360639399813L)
Difference(1)
Difference(2)
Difference(2,193562225603L)
Difference(2,641548912186L)
Difference(2,360639399813L)
Difference(4)
Difference(4,193562225603L)
Difference(4,641548912186L)
Difference(4,360639399813L)
Difference(8)
Difference(8,193562225603L)
Difference(8,641548912186L)
Difference(8,360639399813L)