Intel® QuickAssist Technology (Intel® QAT): Using Adler-32 Checksum and CRC32 Hash to Ensure Data Compression Integrity

White Paper

April 2018
Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and Xeon are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2018, Intel Corporation. All Rights Reserved.
Table 4. Full Empirical Results .......................................................... 21
Table 5. Empirical Study Results ...................................................... 22
Table 6. Worst Case Scenario .......................................................... 22
## Revision History

<table>
<thead>
<tr>
<th>Document Number</th>
<th>Revision Number</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>337373</td>
<td>001</td>
<td>Initial release.</td>
<td>April 2018</td>
</tr>
</tbody>
</table>

§
1 Overview

Data integrity practices refer to the methods used to ensure a digital copy (data set) is identical to the original. Data compression integrity deals with the ability to ensure that a compressed data set can be decompressed at a later time and that decompressed data set will be identical to the original data (before compression).

This paper focuses on the use of both Adler-32 (a checksum algorithm) and CRC32 (a hash algorithm) as a method to determine if the decompressed data set matches the original data set. We show that the Adler-32 checksum and the CRC32 hash have no correlation in the methods used to calculate their respective checksum and hash. Thus, an empirical examination of Adler-32 collision and CRC32 collision can lead to a determination of data integrity with a high degree of confidence.

1.1 Audience

Networking and storage infrastructure architects who want an in-depth understanding of the Intel® QuickAssist Technology (Intel® QAT) Compress and Verify feature to support the planning of reliability, accessibility, and serviceability features in the server environment.
Intel has introduced the Compress and Verify feature of Intel® QAT, which ensures data compression integrity. And this paper discusses the implementation of the Verify component of this feature, which uses Adler-32 checksum and CRC32 hash as a detection mechanism to ensure data compression integrity.

To ensure that compressed data has retained its integrity, an application can immediately decompress the data and test for correctness. If correctness has been established, the compressed data can be safely stored and correctly decompressed later. Alternatively, if the compressed data cannot be correctly decompressed, other remedial actions can be taken immediately (preferable to discovering incorrect data at a later decompression time when the original data may not be available).

**Note:** Correctness is established by comparing the Adler-32 checksum and the CRC32 hash values of the original (uncompressed data) with the decompressed data.

Adler-32 checksum and CRC32 hash are chosen for this study as they are commonly used in the zlib* block and gzip* block footers, respectively. Implementations of the checksum and hash exist for many compressors and decompressors that support the standard deflate block. At the heart of the question of effectiveness of using checksum and hash values is collisions (also known as aliasing). All algorithms that execute a transformation of multiple data sets to checksum or hash will eventually suffer collisions if the originating data set is larger than the size of the checksum or hash.

For example, if the originating data set is a four kilobyte buffer and the algorithm is a standard CRC32 hash, at some point two different input buffers will generate the same CRC32 hash.

We will show that there is no correlation between the construction of Adler-32 checksums and CRC32 hash. Showing that there is no correlation allows the likelihood of collisions to be analyzed independently. Results of an empirical experiment to simulate loss of data integrity (combined with the number of Adler-32 and CRC32 collisions between the original data set and the corrupted data set) can be found in Appendix A and Appendix B.
We discuss the effectiveness of using the Adler-32 checksum and CRC32 hash together as a detection mechanism for ensuring data compression integrity. And we address the likelihood of simultaneous Adler-32 and CRC32 collisions in corrupted data sets. The discussion will be centered on common buffer lengths for compression workloads and references will be made to data set composition (where applicable).

This paper will not address the probability of a loss of integrity, nor the probability that a corrupted data set would correctly decompress, creating a data set with the same length as the original data set. It does include examples of why corruption introduced by the compressor tends to cause a larger impact on the decompressed data.
4 How we Define Data Compression Integrity

Data compression integrity as part of the Intel® QAT Compress and Verify feature is defined as the logical AND of four terms:

\[ val = (rc_d == OK) \&\& (len_d == len_s) \&\& (Adler32_d == Adler32_s) \&\& (crc32_d == crc32_s) \]

Table 1. Description of Parameters for Data Compression Integrity

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rc_d )</td>
<td>Indication of whether or not the decompression operation indicated success</td>
</tr>
<tr>
<td>( len_s )</td>
<td>Clear text length of the original clear text source data</td>
</tr>
<tr>
<td>( len_d )</td>
<td>Clear text length of decompressed data</td>
</tr>
<tr>
<td>( Adler32_s )</td>
<td>Adler-32 checksum calculated across the source data</td>
</tr>
<tr>
<td>( Adler32_d )</td>
<td>Adler-32 checksum calculated across the decompressed data</td>
</tr>
<tr>
<td>( crc32_s )</td>
<td>Standard 32-bit CRC hash calculated across the source data.</td>
</tr>
<tr>
<td>( crc32_d )</td>
<td>Standard 32-bit CRC hash calculated across the decompressed data.</td>
</tr>
</tbody>
</table>

It's expected that most compression errors would be detected by an error during the decompression operation \((rc_d == OK)\) or a decompression operation that results in an incorrect length \((len_d == len_s)\). An error in compression that still results in a data set that decompresses without error to a different data set of the same length is highly unlikely. By its nature, the act of compressing a data set reduces multiple identical patterns of data to one data pattern, and then references to the original for the other patterns.
4.1 Example

The following

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>012345678901234567890123456789012345678901234567890</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I don't know half of you half as well as I should like; and I

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2345678901234567890123456789012345678901234567890</td>
<td></td>
</tr>
</tbody>
</table>

like less than half of you half as well as you deserve.

can be represented as:

I don't know half of you[12,6]as well[8,4]I should like; and{19,3}[12,4] less than{64,28}[20,5]deserve.

During compression, if any of the original characters in “half” become corrupted, the action of copying those characters to the data set to satisfy [distance, length] notations will propagate the corruption multiple times. In the above example, this would result in three errors.

Similarly, if a single length value is misrepresented, then a copy operation will be of the wrong size and the rest of the decompressed data will be the wrong size. If all length values of a specific length are encoded incorrectly, then the problem compounds. In the example above, if a logical error existed to cause all instances of the length [n,4] to be encoded incorrectly, then the overall length of the decompressed data would be incorrect due to two usages of the length 4. The same argument holds for distances.

4.2 Correct Length but Invalid Data

It is hypothetically possible for a compressor to misbehave and create a file with the same number of bytes that it successfully decompresses. If the compressor produces the bits for a different, valid literal (for example, the representation of the letter 'A'), and if the encoding for both literals (the literal that was encoded vs. the literal the should have been encoded) are the same length, then a decompressor would be able to successfully decompress the compressed data set, and it would be the same length. The same would apply to the output of bits for a different, valid distance symbol of the same length.
4.2.1 Example: Static Compression Mismatch

During a compress operation to create an RFC compliant deflate block with static Huffman trees, it is relatively easy to construct a deflate block that, when decompressed, would produce valid data of the same length. From RFC 1951, the static encoding for two literals are shown below:

<table>
<thead>
<tr>
<th>Literal</th>
<th>ASCII hex</th>
<th>Static encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>0x41</td>
<td>0111 0001</td>
</tr>
<tr>
<td>'m'</td>
<td>0x6d</td>
<td>0110 1101</td>
</tr>
</tbody>
</table>


If the compressor outputs the bits for the letter 'm' instead of the letter 'A', the resulting compressed data could be successfully decompressed and would result in a data set of the same length as the original.

This example extends to dynamic compression and to distance tokens. The example can be generalized as follows:

If a compressor does not output a correct literal or distance token, substituting it for a different, incorrect literal or distance code that is the same length and valid, then the resulting deflate block will be valid, per the standards, and will decompress without error. And the result of the decompression operation will be incorrect.
5 Adler-32 Checksum

Adler-32, a 32-bit character-by-character checksum, is a computationally straightforward mechanism for detecting errors after decompression.

5.1 Checksum Method

The formula for Adler-32 can be represented by the following code segment:

```c
const uint32_t MOD_ADLER = 65521;
/*
   65521 is the largest prime number less than 2^16
*/

uint32_t adler32(unsigned char *data, size_t len)
/*
   where data is the location of the data in physical memory and
   len is the length of the data in bytes
*/
{
    uint32_t a = 1, b = 0;
    size_t index;

    // Process each byte of the data in order
    for (index = 0; index < len; ++index)
    {
        a = (a + data[index]) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }
    return (b << 16) | a;
}
```

Source: https://en.wikipedia.org/wiki/Adler-32
5.1.1 Example

The Adler-32 checksum of the ASCII string "QuickAssist" would be calculated as follows:

Table 3. Adler-32 Checksum of an ASCII String

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
<th>a (init=1)</th>
<th>b (init=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>prev a +</td>
<td>data</td>
<td>new a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(hex)</td>
</tr>
<tr>
<td>Q</td>
<td>81</td>
<td>1</td>
<td>81</td>
</tr>
<tr>
<td>u</td>
<td>117</td>
<td>82</td>
<td>117</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
<td>199</td>
<td>105</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
<td>304</td>
<td>99</td>
</tr>
<tr>
<td>k</td>
<td>107</td>
<td>403</td>
<td>107</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
<td>510</td>
<td>65</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>575</td>
<td>115</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>690</td>
<td>115</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
<td>805</td>
<td>105</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>910</td>
<td>115</td>
</tr>
<tr>
<td>t</td>
<td>116</td>
<td>1025</td>
<td>116</td>
</tr>
</tbody>
</table>

\[ a = 1141 = 0x0475 \text{ (base 16)}; \quad b = 6644 = 0x19F4 \]

Output (Checksum) = 0x19F4 \ll 16 + 0x0475 = 0x19F40475

In this example, the upper 16 bits of the Adler-32 checksum are the 'b' value, 6644 (decimal), and the lower 16 bits are the 'a' value, 1141 (decimal). Converting to hexadecimal, the entire checksum (output) is 0x19F40475.

Note: The modulo operation had no effect in this example, as none of the values reached 65521.

5.2 Collisions

5.2.1 Character Sets and Collisions

For an Adler-32 checksum collision to occur, both modulo operations must occur. Until both of these events happen, the checksum will be unique with respect to previous Adler-32 calculations on the same buffer. And any change in the buffer due to an error
will result in a different checksum. The rate of modulo operations occurring during calculation can be interpreted as representative of the rate of potential collisions.

5.2.2 **ASCII Values**

5.2.2.1 **Lower (0-85)**

If the bytes of data provided to the Adler-32 function are at the lower end of the ASCII character set, the rate of the modulo operations occurring will be less than if the data is in the midrange or higher end of the ASCII table. If an error is introduced into the data, the lower modulation rate will mean the error is less likely to collide with the original data.

5.2.2.2 **Mid-range (86-170)**

If the data is predominantly in the middle range of the ASCII character set, the rate of collisions is highest. This is an artifact of both higher modulation rates, and higher likelihood of a three-byte change collision, or a variant.

5.2.2.3 **High (170-255)**

If the bytes of data are at the higher (upper) end of the ASCII character set, then there will be more frequent modulation. However, because larger ASCII values are used, it is less likely the error will induce the three-byte change collision, or a variant, in the checksum.

See Appendix A and Appendix B for an empirical analysis of these statements.

5.2.3 **Three-Byte Change Collisions**

The most commonly studied collisions in the Adler-32 checksum involve a change of three consecutive bytes. The collision is derived from studying the algorithm itself. Since the algorithm itself is built on additions, then it is a simple matter of selecting three bytes whose relative values will result in a zero-sum difference to the Adler-32 checksum itself.

This has been researched and any multiple of the vector \(\{1, -2, 1\}\) will satisfy. For example, assume a character array ends with the letters "...234", and it has the Adler-32 "A". If those last three character are changed to "...305", then the Adler-32 will still be the same, assuming no modulation occurs.

*Source: [http://www.leviathansecurity.com/blog/analysis-of-adler32](http://www.leviathansecurity.com/blog/analysis-of-adler32)*
5.3 Non-Collisions

The Adler-32 checksum has the following known properties:

1. All single bit flips will be detected.
2. All double bit flips will be detected.
3. Burst errors up to seven bits are always detected.

Source: https://www.zlib.net/maxino06_fletcher-adler.pdf
The hash function used to calculate the check-value for the gzip footer is a 32-bit CRC with the $0xebd88320$ polynomial.

CRC32 is calculated by a polynomial division and the data buffer is divided by the polynomial. The remainder is the CRC32. A concise example can be studied at the referenced source below.

Source: [http://cs.newpaltz.edu/~easwaran/CN/Module7/CRC2.pdf](http://cs.newpaltz.edu/~easwaran/CN/Module7/CRC2.pdf)

6.1 Bit Changes in the Hash

CRC (Cyclic Redundancy Check) functions are designed to provide an even distribution of hash values. The mechanism used to provide the even distribution is division, with the remainder becoming the hash value.

The division used in the CRC function is an XOR operation (instead of a subtraction) and leading zeroes are dropped. Also, the use of remainders in subsequent division operations has the effect of making changes in multiple locations of the final CRC.

6.1.1 Example:

If the divisor is $110011$, and the dividend is $1100101010011$, then the CRC would be:

```
110011 | 110010101001100000  \rightarrow dividend lengthened by 5 bits for CRC  
  110011
  -------
    1101001100000
    11011
    -------
      101100000
      11011
      -------
        11010000
        110011
        -------
          11100  \rightarrow remainder is CRC
```
If one bit is changed in the dividend:

```
  110011  |110  1010 1001 1000 00
                  1100 11
  ----------------
    1  0110 1001 1000 00
   1  1001  1
  ---------------
     1111  0001 1000 00
    1100 11
  ---------------
       11  1101 1000 00
      11  0011
  ---------------
        1110 1000 00
       1100 11
  ---------------
         10  0100 00
        11  0011
  ---------------
          1  0111 00
         1  1001 1
  ---------------
            1110 10
           1100 11
  ---------------
             010 01  ← remainder is the CRC
```

The example above shows that one changed bit changes three of the bits in the 5-bit CRC. And this very important concept indicates that localized changes in the data buffer are reflected with non-localized changes in the CRC.

## 6.2 Byte-by-byte

Due to the nature of the type of algorithms being implemented, it's usually implemented as a lookup table of 256 remainders. See the source referenced below for details and a concise explanation.

**Source:** https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code

The effect of bit changes on the CRC32 can be clearly seen with a standard table based implementation. The standard CRC32 with the 0xedb88320 polynomial can be implemented with a table of remainders. Select values are shown below:
For select values:

'x'  \text{crc_table}[0x78] = 0x5edef90e = \\
0101111011110111110010011110 \\
'y'  \text{crc_table}[0x79] = 0x29d9c998 = \\
00101001110101001100110000

As can be seen, if a single bit is changed, for example `x` (0x78) to `y` (0x79), then the cumulative calculation of the CRC32 is impacted by 15 bit changes, spread across the length of the remainder.

For select values:

'M'  \text{crc_table}[0x4d] = 0x86d3d2d \\
'N'  \text{crc_table}[0x4e] = 0x91646c97 \\
'O'  \text{crc_table}[0x4f] = 0xe6635c01

The same mixing of hash values, relative to the number of changed bits, is observed.

See Appendix C for example source code to create the remainder table.

6.3 Non-Collisions

The industry standard CRC32 with the 0xedb88320 polynomial has the following known properties:

1. All single bit flips will be detected.
2. All double bit flips will be detected.
3. All 32-bit bursts of errors will be detected.
4. The CRC32 has a hamming distance of 4 for data lengths up to 91706 bytes.
5. A hamming distance of 5 for data lengths up to 2974 bytes.

\§
It can be concluded from examining the implementation of the Adler-32 checksum and the CRC32 hash that the values are uncorrelated. The Adler-32 checksum is derived from modulated additions and the CRC32 hash is derived from integer division (with remainders).

Empirical studies were performed to measure the collision rate of Adler-32, with the worst case being one in 52 million. Empirically, the worst case measured for the CRC32 collision was one in 1.04 billion. All data corruptions in the study that resulted in a CRC32 collision were detected by the Adler-32 checksum, and all data corruptions that resulted in an Adler-32 collision were detected by the CRC32 hash. The observed rate of both an Adler-32 collision and CRC32 collision on the same data was 0 in 1.5 trillion corruptions.

In conclusion, testing for data compression integrity using Adler-32 checksum and CRC32 hash together provides increased likelihood of detecting corruption when compared to using either in isolation. See Appendix A for a complete test description and Appendix B for complete test results.
Appendix A Empirical Study Description

Three data set sizes of 4, 8, and 32 kilobytes were selected and the empirical study was conducted as follows:

For each data set size

Five data set content ranges were selected:

1. Low ASCII values (0-85)
2. Mid-range ASCII values (86-170)
3. High ASCII values (170-255)
4. The set of printable ASCII values (32-127)
5. The full ASCII set (0-255)

For each data set range

- Thirty-two different clean_buffers were created by random selection of ASCII characters within the data set range.
- The Adler-32 and CRC32 of the clean_buffer were calculated.

For each clean_buffer 3,125,000,000 loops

- A corrupted_buffer was created by replacing a random number of up to 100 characters. The location in the buffer was randomized from start to end, and the replacement character was randomized from the full set of ASCII values.
- The Adler-32 checksum was calculated, and compared with the Adler-32 from the clean_buffer. If these checksums were the same, a collision was recorded.
- The CRC32 hash was calculated, and compared with the CRC32 from the clean_buffer. If these hash values were the same, a collision was recorded.

For each data set size and data set range, a total of 100,000,000,000 corrupted buffers were constructed and the checksum and hash were compared. This number was chosen to represent more than an order of magnitude greater than the optimal expected collision rate of 1 in 4,294,967,296 for a 32-bit checksum or hash.

Note: The standard Linux* rand() function seeded with the time() function was used to generate random values.
### Table 4. Full Empirical Results

<table>
<thead>
<tr>
<th>Input Data size (bytes)</th>
<th>Location in ASCII Table</th>
<th>CRC32 Collisions (in 100 billion corrupted data sets)</th>
<th>Adler-32 collisions (in 100 billion corrupted data sets)</th>
<th>Double collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>Low</td>
<td>29</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>Med</td>
<td>16</td>
<td>1303</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>High</td>
<td>18</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>Printable ASCII</td>
<td>30</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>Full/(Unicode)</td>
<td>24</td>
<td>942</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>Low</td>
<td>27</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>Med</td>
<td>24</td>
<td>1340</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>High</td>
<td>21</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>Printable ASCII</td>
<td>31</td>
<td>115</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>Full</td>
<td>20</td>
<td>965</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>Low</td>
<td>20</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>Med</td>
<td>25</td>
<td>1299</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>High</td>
<td>24</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>Printable ASCII</td>
<td>29</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>Full</td>
<td>21</td>
<td>606</td>
<td>0</td>
</tr>
</tbody>
</table>
### 7.1 Summary

In summary, the empirical study generated 480 buffers, and then corrupted each buffer 3,125,000,000 times for a total of 1.5 trillion corruptions. The Adler-32 and CRC32 of the original clean buffer and the corrupted buffer were compared and the number of collisions are summarized in the table below. In every corrupted data set, either the CRC32 or the Adler-32 or both were different from that of the clean buffer (refer to the double collisions in the tables below). No corrupted buffer was not detected by examining the Adler-32 and CRC32.

#### Table 5. Empirical Study Results

<table>
<thead>
<tr>
<th></th>
<th>CRC32</th>
<th>Adler-32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Collisions (in 1.5 trillion corruptions)</strong></td>
<td>359</td>
<td>6877</td>
</tr>
<tr>
<td><strong>One in (approximately):</strong></td>
<td>4,178,272,980</td>
<td>218,118,365</td>
</tr>
<tr>
<td><strong>Double Collisions</strong></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

As previously noted in this paper, the likelihood of collision by Adler-32 changes is based on the data set in use. The largest number of collisions for the Adler-32 checksum was 59. This was observed in the 8 kilobyte dataset with the mid-range ASCII data set. Independently, there were several tests that showed three CRC32 collisions out of 3,125,000,000 corruptions. This worst case scenario is summarized below.

#### Table 6. Worst Case Scenario

<table>
<thead>
<tr>
<th></th>
<th>CRC32</th>
<th>Adler-32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collisions (in 3.125 billion corruptions)</strong></td>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td><strong>one in (approximately):</strong></td>
<td>1,041,666,667</td>
<td>52,966,101</td>
</tr>
<tr>
<td><strong>Double Collisions</strong></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
7.2 View by ASCII Data Set

Figure 1. 96 Samples

Collisions per 3.125 billion corruptions
(96 samples each)

7.3 View by Data Set Size

Figure 2. 160 Samples

Collisions per 3.125 billion corruptions
(160 samples each)
Appendix C  Code to Create full CRC32 Table


/* Table of CRCs of all 8-bit messages. */
unsigned long crc_table[256];

/* Flag: has the table been computed? Initially false. */
int crc_table_computed = 0;

/* Make the table for a fast CRC. */
void make_crc_table(void)
{
    unsigned long c;
    int n, k;
    for (n = 0; n < 256; n++) {
        c = (unsigned long) n;
        for (k = 0; k < 8; k++) {
            if (c & 1) {
                c = 0xedb88320L ^ (c >> 1);
            } else {
                c = c >> 1;
            }
        }
        crc_table[n] = c;
    }
    crc_table_computed = 1;
}
/*
   Update a running crc with the bytes buf[0..len-1] and return the updated crc. The crc should be initialized to zero. Pre- and post-conditioning (one's complement) is performed within this function so it shouldn't be done by the caller. Usage example:
   unsigned long crc = 0L;
   while (read_buffer(buffer, length) != EOF) {
      crc = update_crc(crc, buffer, length);
   }
   if (crc != original_crc) error();
*/
unsigned long update_crc(unsigned long crc,
    unsigned char *buf, int len)
{
    unsigned long c = crc ^ 0xffffffffL;
    int n;

    if (!crc_table_computed)
        make_crc_table();
    for (n = 0; n < len; n++) {
        c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
    }
    return c ^ 0xffffffffL;
}

/* Return the CRC of the bytes buf[0..len-1]. */
unsigned long crc(unsigned char *buf, int len)
{
    return update_crc(0L, buf, len);
}