SELECTION OF HASHING ALGORITHMS

Tim Boland
Gary Fisher

JUNE 30, 2000

INTRODUCTION

The National Software Reference Library (NSRL) Reference Data Set (RDS) is built on file signature generation technology that is used primarily in cryptography. The selection of the specific file signature generation routines is based on customer requirements and the necessity to provide a level of confidence in the reference data that will allow it to be used in the U.S. Courts. This document gives an overview of the various hashing algorithms considered, as well as implementations of those algorithms. It also gives factors regarding their selection and use.

Hashing is an extremely good way to verify the integrity of a sequence of data bits (e.g., to make sure the contents of the sequence haven't been changed inadvertently). The sequence might make up a character string, a file, a directory, or a message representing data (binary 1s or 0s) stored in a computer system. The word "hash" means to "chop into small pieces" (REF1). A hashing algorithm is a mathematical function (or a series of functions) taking as input the aforementioned sequence of bits and generating as output a code (value) produced from the data bits and possibly including both code and data bits. Two files with exactly the same bit patterns should hash to the same code using the same hashing algorithm. If a hash for a file stays the same, there is only an extremely small probability that the file has been changed. On the other hand, if the hashes for the files do not match, then the files are not the same. Thus, hashes could be used as a primary verification tool to find identical files. The output code of the hash function should have a "random" property, so that different sequences of bits hash to different values as much as possible. Hashes are used in "scatter storage" systems, in digital signature applications, and recently in computer forensics applications, to determine whether the contents of a suspect machine have been modified maliciously. Hashing algorithms can be efficiently implemented on modern computers.

Hashing algorithms fall within the realm of error detection techniques. In a general sense, the aim of an error detection technique is to enable the receiver of a message transmitted through a noisy (error-introducing) channel to determine whether the message has been corrupted. Some hashing algorithms perform complex transformations on the message to inject it with redundant information, while others leave the data intact and append a hash value on the end of a message. In any case, the transmitter may construct a hash value that is a function of the message. The receiver can then use the same hashing algorithm to compute the hash value of the received message and compare it with the transmitted hash value to see if the message was correctly received. If the hash values match, then the

message was correctly received; if not, then there must have been an error in one or more of the data bits of the message.

The National Institute of Standards and Technology (NIST) of the U.S. Department of Commerce has been asked to investigate commonly-used hashing algorithms in support of the National Software Reference Library (NSRL). There are several algorithms available, differing in complexity, robustness, ease of use, and machine efficiency. Generally, “hardware (physical device)” methods of computing hashes involve extensive bit manipulations, and are relatively inefficient for a software (programmatic) computation, so the algorithms discussed contain software methodologies to streamline the hashing process. What is involved in all the algorithms is a method to break up the input into manageable portions, and manipulate the input in a systematic way over and over (iteratively). The algorithms generally differ in the degree to which they do this, and the number of iterations involved.

Given the above, NIST investigated available implementations of four different hashing algorithms and tested the algorithm output on some test data. The purpose of this exercise was to define some “reference” implementations for verifying the correctness of entries in the NSRL Reference Data Set (RDS). Multiple algorithms were considered because of the need for “double-checking” results and because many facilities use multiple hashing algorithms simultaneously. Performance metrics mentioned above were used in evaluating candidate implementations.

For each algorithm, the authorities (official sources and sanctions) of the source programs and tests used for testing accuracy will be described. Algorithms mentioned in this report may have limitations (clashes found, performance, etc.), which will be mentioned as appropriate. All implementations evaluated are freely available from the Internet. Each of the four algorithms will be described below, starting with a high-level overview and progressing to more detail as appropriate.

CRC32

The cyclic redundancy code (CRC) algorithm is the simplest of the four hashing algorithm choices, but also the least robust. The name means that the algorithm operates in repetitive (cyclic) redundant cycles to produce an output hash code. The “32” indicates the number of bits being considered to produce the hash code (explained below). The CRC algorithm is a key component in the error-detecting capabilities of many communications protocols. In a CRC algorithm, the transmitter of a message constructs a value (called the checksum) and appends it to the message. The receiver can then use the same function to compute the checksum of the received message and compare it with the appended checksum to see if the message was correctly received. For example, if we chose a checksum function which was the sum of the decimal numbers in a message, it might go something as follows: Message-1 2 3, Message with checksum – 1 2 3 6 (6 is sum of 1 and 2 and 3), Message after transmission – 1 2 4 6. Here the third decimal number was corrupted from 3 to 4, and the receiver can detect this by computing the checksum (7=1+2+4) from the message, and compare it with the transmitted checksum of 6. Obviously, both sender and receiver must be using the same algorithm to be consistent.

If the checksum itself is corrupted, a correctly transmitted message might be incorrectly identified as a corrupted one. However, this is a side-safe failure. A dangerous-side failure occurs where the message and/or checksum is corrupted in a manner that results in a transmission that is internally consistent. Unfortunately, this possibility is completely unavoidable and the best that can be done is to minimize its probability by increasing the amount of information in the checksum (REF2).

The above example is obviously very simple, and would not suffice for rigorous error detection. A more complex checksum function is needed. While addition is clearly not strong enough to form an effective checksum, it turns out that division is, so long as the divisor (number to divide by) is about as wide as the checksum register (place to store the checksum value).

The basic idea behind CRC algorithms is simply to treat the message as an enormous binary number, to divide it by another fixed binary number, get a quotient, and make the remainder from this division the checksum. Upon receipt of the message, the receiver can perform the same division and compare the remainder with the "checksum" (transmitted remainder). For example, when dividing decimal 11 (message) by 4 (divisor) we get a value of 2 (quotient) with a remainder of 3.

With CRC division, instead of viewing the numbers mentioned above as positive integers, they are viewed as polynomials with binary coefficients. This is done by treating each number as a bit-string whose bits are the coefficients of a polynomial. For example, the ordinary number 23 (decimal) is 10111 (binary) and so it corresponds to the polynomial x**4 + x**2 + x**1 + x**0. Polynomials are used because they provide useful mathematical machinery in the calculations. CRC arithmetic is primarily about XORing (exclusive-ORing) particular values at various shifting offsets, which has the effect of doing the binary division. An exclusive-OR function produces 1 if the two input bits are different; otherwise it produces 0.

The CRC algorithm can be applied to messages of different widths (12, 16, or 32 bits). We are considering the 32-bit (CRC32) algorithm here because it is the most robust. In this case the polynomial is 32 bits wide and the CRC32 checksum is also 32 bits. This also simplifies the calculation on most modern computers. Other CRC polynomials used besides CRC32 are CRC12, CRC16, and CRC-CCITT, from the Consultative Committee for Telephone and Telegraph (CCITT).

On PCs one can deal with binary numbers of only 32 bits or fewer, so one must break up the enormous binary number mentioned above into manageable chunks. That's exactly what the two CRC algorithms mentioned below do. In order to speed up the process, the algorithms use a pre-calculated look-up table; the table contains a CRC for each character code between 0 and 255, so that the calculation doesn't need to be repeated as the text strings are processed. This process has the effect of performing the division of the enormous binary number by the generator polynomial, but in increments, due to the limitations of modern computing. In other words, instead of computing the CRC bit by

bit, a 256-element lookup table can be used to perform the equivalent of 8 bit operations at a time.

To perform a CRC calculation, the user needs to choose a divisor. Generally the divisor is called the “generator polynomial” or simply the “polynomial”, and is a key parameter of any CRC algorithm. One can choose any polynomial and come up with a CRC algorithm. However, some polynomials are better then others. An example of a polynomial used might be 79,764,919 decimal, or 0x04c11db7 hexadecimal. Theoretical mathematicians have calculated certain polynomials to provide the least duplications in remainders.

CRC Implementation

To implement a CRC algorithm is to implement CRC division. There are two reasons why the divide instruction of the host machine cannot be used. The first is that the division must be in CRC arithmetic. The second is that the dividend might be ten megabytes (1 byte=8 bits) long, and today’s processors do not have registers large enough to hold a dividend of this size. To implement CRC division, we have to feed the message in smaller chunks through a division register.

Originally there were seven candidate CRC32 implementations (using C or C++ high-level programming languages) under consideration (which represents nearly all of the researched CRC32 implementations publicly available). Performance metrics used to evaluate these implementations were the following (in no particular order of importance): speed of execution, ease of use, accuracy, ability to operate on entire files, and choice of generator polynomial. One implementation was rejected because it did not produce accurate results, two were not set up to operate on entire files (only text strings), and two were slow (because they were not “table-driven”). Only two were reasonably fast, produced accurate information, were table driven, and used generally accepted generator polynomials. Both are table-driven, but one uses a polynomial is from an American National Standards Institute (ANSI) X3 Committee, while the other polynomial is not explicitly specified in code, but the table entries are the same as compared to the other. The two implementations are about the same number of programming statements. Slight preference was given for the algorithm that computes values for directories of files as well as individual files.

The test data used to verify the routines was from commonly used PKZIP (REF3) and WINZIP (REF4) products, and other various test character strings and file directories. Since these products are commonly used and routinely generate CRCs, they would be valid benchmarks of accuracy. The CRC outputs are in hex. Both implementations verified correctly against the data. There are no apparent limitations in the implementations, other than the inherent CRC32 limitations, although one implementation produces more cursory output on only one file at a time. More information on each of these implementations is given below.

The first candidate CRC program (using the C language) computes the 32-bit CRC used as the frame check (error-detection) sequence in FIPS 71 (REF5) This source code is from the Snippets file collection (REF6). It consists of a header file, crc.h, and a main

program crc_32.c. For this driver routine, first the polynomial itself and its table of feedback terms is provided. The polynomial is:

*x**32 + x**26 + x**23 + x**22 + x**16 + x**12 + x**11 + x**10 + x**8 + x**7 + x**5 + x**4 + x**2 + x**1 + x**0.*

The polynomial is taken backwards and the highest-order term is placed in the lowest-order bit. The x**32 term is "implied; the least significant bit is the x**31 term, etc. The x**0 term (usually shown as "+1") results in the most significant bit being 1. A hardware shift register implementation shifts bits into the lowest-order term (to the right). It is optimized here by shifting eight-bit chunks at a time. The calculated CRC must be transmitted in order from highest-order term to lowest-order term. The feedback terms table consists of 256 32-bit entries. The feedback terms represent the results of eight shift/XOR operations for all combinations of data and CRC register values. The CRC accumulation logic is the same for all CRC polynomials; the appropriate table just needs to be chosen. The table can also be generated at runtime. The values must be right-shifted by eight bits by the logic in the *updateCRC* routine called from the main program; the shift must be unsigned (masked with zeroes in the high-order bits). On some hardware the shift could probably be optimized by using byte-swap instructions. Unsigned variables need to be used consistently.

The second candidate CRC program (also using the C language) computes a composite CRC that is not dependent on the endian type of the machine executing the program. "Endian" refers to the order in which received bits are stored. This means the composite CRC-32 can be used to test the transfer of a set of files, when transferred in binary mode, between machines of different architecture. It is adapted from the "charcnt.c" program and "crc16.u" unit modified to include a CRC32 table from Microsoft Systems Journal (MSJ) (REF16). "crc32" gives the same values as the PKZIP utility, and has been verified using (1) Borland C/C++ (REF14) and (2) Sun C++ (REF15). This source code was copyrighted by Earl F. Glynn in 1998 and is available from efg's Computer Lab Mathematics site (REF7). This implementation consists of several files: a driver program crc32.c, and several header files. The driver program first defines a table used for byte-wise calculation of CRC32. The routine in the main program executes very quickly as follows: (1) the input byte is XORed with the low-order byte of the CRC "register" to get an index into the table, (2) the CRC "register" is shifted eight bits to the right, and (3) the CRC register is XORed with the contents of Table[Index]. Steps (1) through (3) are executed for all input bytes. The result in the CRC register is the CRC.

In sum, CRC hashes are the simplest of those considered, but are also the weakest, in that CRC values can be compromised in terms of verification and error detection. (The final decision of which form of the algorithm to use will be based on compatibility with software provided by the customer. The software is still an unknown in this equation.)

MD4

MD4 (message digest level 4) is a one-way hash function designed by Ron Rivest. One-way hash functions (see REF8) have certain characteristics, in addition to the

characteristic of taking an arbitrary-length input and returning an output of fixed length; they are able to provide a “fingerprint” of a message that is unique. Thus, the MD4 algorithm takes as input a message of arbitrary length; the algorithm produces as output a 128-bit hash, or message digest, of the input message. MD4 is more complex than CRC32 mentioned above, so it is more “computer-intensive,” but it performs transformations on the data itself instead of just appending a checksum as in CRC32, so it is more robust, and provides a greater verification and error-detection ability at the price of greater complexity. The design goal was that it would be computationally infeasible to find two messages that hashed to the same value, or produced the same message digest. It would also be computationally infeasible to produce any message having a given pre-specified target message digest.

The MD4 message-digest algorithm provides a “fingerprint” of a message of arbitrary length. The difficulty of coming up with two messages having the same message digest is on the order of 2**64 operations, and the difficulty of coming up with any message having a given message digest is on the order of 2**128 operations.

The MD4 algorithm is intended for digital signature applications, where a large file must be “compressed” in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem. This involves disguising the contents of a file so that it may be read only by intended recipients. The MD4 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD4 algorithm does not require any large substitution tables; the algorithm can be coded compactly. MD4’s security is not based on any assumption, such as the difficulty of factoring. MD4 is suitable for high-speed software implementations; it is based on a simple set of bit manipulations on 32-bit operands.

MD4 is as simple as possible, without large data structures or a complicated program, and is optimized for microprocessor architectures. The MD4 implementation (in the C language) chosen was from Internet Engineering Task Force (IETF) RFC1320 (REF9). It was chosen because it faithfully reproduces the MD4 algorithm (also found in RFC1320) and is portable. In summary form, the MD4 algorithm works as follows: (1) padding bits are appended to the file, (2) the length is appended, (3) the MD buffer is initialized, the message is processed in 16-word blocks, and (5) output is produced. A detailed description of these steps follows.

Suppose there is a b-bit message as input, and desired message digest from that input as output. Here b is an arbitrary nonnegative integer, and it may be arbitrarily large. Thus the message may be written down as follows: m_0 m_1 …. M(b-1).

For (1) above, the message is then padded (extended) so that its length (in bits) is just 64 bits shy of being a multiple of 512 bits long. Padding is performed as follows: a single “1” bit is appended to the message, and then “0” bits are appended subject to the requirement above. In all, at least 1 bit and at most 512 bits are appended.

For (2) above, a 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. At this point, the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits (meaning that the message length divided by 512 is an integer). Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words.

For (3) above, a four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, and D is a 32-bit register (high-speed storage unit), initialized as follows: Word A: 01 23 45 67, Word B: 89 ab cd ef, Word C: fe dc ba 98, Word D: 76 54 32 10.

For (4) above, three auxiliary functions (F, G, H) are defined that each take as input three 32-bit words and produce as output one 32-bit word. In each bit position F acts as a conditional: if X then Y else Z. In each bit position G acts as a majority function: if at least two of X, Y, and Z are on, then G has a "1" bit in that bit position; otherwise, G has a "0" bit. The function H is the bit-wise XOR or "parity (error checking)" function; it has properties similar to those of F and G. Then each 16-word block is processed in three rounds as outlined below. Each round uses a different operation 16 times, and each round involves one of the functions F, G, or H (e.g., round 1 uses F, round 2 uses G, and round 3 uses H). Each operation performs a function on three of A, B, C, and D. Then it adds that result to the fourth variable, a sub-block of the text, and a constant (which could be 0). It then rotates that result to the right a variable number of bits. Then the result replaces one of A, B, C, or D. Finally, each of the four registers A, B, C, D mentioned above, is incremented by the values held before processing. This concludes the processing of a 16-word block; then it's time to move on to the next 16-word block until there are none left.

For (5) above, the resultant message digest produced is the values contained in A, B, C, and D. These values are "concatenated", such that the digest starts with the first bit of A and ends with the last bit of D.

In summary, the MD4 algorithm produces a "fingerprint," or message digest of a message of arbitrary length. It has been carefully scrutinized for weaknesses; however, further security analysis is justified. It may be used if a greater degree of error detection than CRC32 is desired.

The MD4 implementation chosen consists of four files: global.h, md4.h, md4c.c, and mddriver.c. The driver compiles for MD5 by default but can compile for MD2 or MD4 if the symbol MD is defined on the C compiler command lines as 2 or 4. The file global.h defines data types and constants. The file md4.h is a header file for md4c.c. In md4c.c, other constants are defined, and then steps (1) through (5) are implemented as described above. The program mddriver.c is a test driver for MD4.

The implementation is portable and should work on many different platforms. It is not difficult to optimize the implementation on particular platforms.

MD4 test results were compared against sample data from the Handbook of Applied Cryptography (REF10), as well as test data in the program from IETF RFC1320, and

were found to be in agreement with test data. There are no known limitations of the implementation other than inherent limitations in the MD4 algorithm. Researchers have shown that clashes, or duplicate hash strings, can be generated from different files.

MD-5

Message Digest level 5 (MD5) is an improved version of MD4. Although more complex than MD4, it is similar in design and also produces a 128-bit hash. After some initial processing, MD5 processes the input text in 512-bit blocks, divided into 16 32-bit sub-blocks. The output of the algorithm is a set of four 32-bit blocks, which concatenate to form a single 128-bit hash value. The MD5 algorithm potentially offers a greater degree of error detection than does MD4, at the price of slightly more complication.

The MD5 message-digest algorithm takes as input a message of arbitrary length and produces as output a 128-bit “fingerprint” or “message digest” of the input. It is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be “compressed” in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem. This involves disguising the contents of a file so that it is recognizable only by intended recipients.

The MD5 algorithm is designed to be very fast on 32-bit machines. In addition, the MD5 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly.

The MD5 algorithm is an extension of the MD4 message-digest algorithm. MD5 is slightly slower than MD4, but is more “conservative” in design. MD5 was designed because it was felt that MD4 was perhaps being adopted for use more quickly than justified by the existing critical review; because MD4 was designed to be exceptionally fast, it was “at the edge” in terms of risking successful cryptanalytic attack (meaning that someone, if they tried enough times, could “break” the code, or recognize a message from its code). MD5 backs off somewhat, giving up a little in speed for a much greater likelihood of ultimate security. It incorporates some suggestions made by various reviewers, and contains additional optimizations.

The MD5 implementation (using the C language) chosen was from IETF RFC1321 (REF11) . It was chosen because it faithfully reproduces the MD4 algorithm (also found in IETF RFC1321) and is portable.

The MD5 message-digest algorithm performs the following five steps: (1) append padding bits, (2) append length, (3) initialize MD buffer, (4) process message in 16-word blocks, and (5) generate output. These steps are described in detail below.

Suppose there is a b-bit message as input, and desired message digest from that input as output. Here b is an arbitrary nonnegative integer, and it may be arbitrarily large. Thus the message may be written down as follows: m_0 m_1 …. M(b-1).

For (1) above, the message is then padded (extended) so that its length (in bits) is just 64 bits shy of being a multiple of 512 bits long. Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended subject to the requirement above. In all, at least 1 bit and at most 512 bits are appended

For (2) above, a 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. At this point, the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits (meaning that the message length divided by 512 is an integer). Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words.

For (3) above, a four-word buffer (A, B, C, and D) is used to compute the message digest. Here each of A, B, C, and D is a 32-bit register (high-speed storage unit), initialized as follows:

Word A: 01 23 45 67, Word B: 89 ab cd ef, Word C: fe dc ba 98, Word D: 76 54 32 10.

For (4) above, four auxiliary functions (F, G, H, and I) are defined such that each takes as input three 32-bit words and produces as output one 32-bit word. In each bit position, F acts as a conditional: if X then Y else Z. The functions G, H, and I are similar to the function F.

The function H is the bit-wise XOR(exclusive-OR, or "parity") function of its inputs. This step uses a 64-element table constructed from the sine trigonometric function. Then each 16-word block is processed in four rounds as outlined below. Each round involves using a different operation 16 times, and each round involves one of the functions F, G, H, or I, plus the table T (e.g., round 1 uses F and T, round 2 uses G and T, round 3 uses H and T, and round 4 uses I and T). Each operation performs a nonlinear function on three of A, B, C, and D. Then it adds that result to the fourth variable, a sub-block of the text and a constant. It then rotates that result to the right a variable number of bits and adds the result to one of A, B, C, or D. The result replaces one of A, B, C, or D. Finally, each of the four registers A, B, C, and D is incremented by the values held before processing. This concludes the processing of the 16-word block, and it's time to move to the next one until there are none left.

For (5) above, the resultant message digest produced is A, B, C, D. These values are "concatenated" such that the digest starts with the first bit of A and ends with the last bit of D.

The following are the differences between MD4 and MD5: (1) a fourth round has been added in MD5; (2) each step in MD5 has a unique additive constant; (3) the function G in round 2 was changed to make G less symmetric (balanced); (4) each step in MD5 adds in the result of the previous step (promoting a faster "avalanche effect"); (5) the order in which input words are accessed in rounds 2 and 3 is changed, to make these patterns less like each other; and (6) the shift amounts in each round have been approximately

optimized to yield a faster “avalanche effect”, and the shifts in different rounds are distinct.

The MD5 message-digest algorithm is relatively simple to implement, and provides a “fingerprint” or message digest of a message of arbitrary length. The difficulty of coming up with two messages having the same message digest is on the order of 2^{64} operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^{128} operations.

The MD5 implementation chosen consists of four files: global.h, md5.h, md5c.c, and mddriver.c, The file global.h defines common data types and constants. The file md5.h is a header file for md5c.c. The file md5c.c defines other constants and then performs the processing defined in steps (1) through (5) above. The file mddriver.c is a test driver for MD5. The implementation is portable and should work on many different platforms. It is not difficult to optimize the implementation on particular platforms.

MD5 test results were compared against sample data from the Handbook of Applied Cryptography, as well as test data in the program from IETF RFC1321, and were found to be in agreement with this test data.. There are no known limitations of the implementation other than the inherent limitations of the MD5 algorithm. The MD5 algorithm has been susceptible to sustained attacks in the past, but it still is the most robust choice so far.

SHA-1

NIST, along with the National Security Agency (NSA), designed the Secure Hash Algorithm Revision 1 (SHA-1) for use with the Digital Signature Standard (DSS) (REF12); this standard is the Secure Hash Standard; SHA-1 is the algorithm used in the standard. Additionally, for applications not requiring a digital signature, the SHA-1 can be used whenever a secure hash algorithm is required. The SHA-1 is specified by NIST Federal Information Processing Standard (FIPS) 180-1 (REF13). The SHA-1 is a technical revision of the SHA (specified by FIPS 180, which has been superseded). The SHA-1 can be used to generate a condensed representation of a message called a message digest. The SHA-1 is used by both the transmitter and intended receiver of a message in computing and verifying a digital signature.

When a message of any length less than 2^{64} bits is input, the SHA-1 produces a 160-bit output called a message digest; this output is longer than that of MD5.. The message digest is usually much smaller in size than the message. The SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with a very high probability, result in a different message digest. The SHA-1 is based on principles similar to those used in the development of MD4, and is closely modeled after MD4. The SHA-1 may be implemented in software, firmware, hardware, or any combination thereof. Implementations of the SHA-1 may be validated by NIST in accordance with a reference

implementation produced at NIST. It is this reference implementation that is selected in this study.

In summary, the SHA-1 is more complex than the choices considered thus far, but presents a more robust solution than the other hashing algorithms considered. Thus far the SHA-1 has been impervious to compromise.

Input to the SHA-1 should be considered to be a bit string. The length of the message is the number of bits in the message. The purpose of message padding is to make the total length of a padded message a multiple of 512. The SHA-1 sequentially processes blocks of 512 bits when computing the message digest. A sequence of logical functions f(0), f(1), … , f(79) is used in the SHA-1. The message digest is computed using the final padded message. The computation uses two buffers, each consisting of five 32-bit words, and a sequence of eighty 32-bit words. A more detailed description of the SHA-1 algorithm is given below.

First, the message is padded to make it a multiple of 512 bits long. Padding is exactly the same as in MD5: first append a one, then as many zeros as necessary to make it 64 bits short of a multiple of 512, and finally a 64-bit representation of the length of the message before padding. Then five 32-bit variables (in contrast to the four for MD5) are initialized as follows (in hexadecimal):

> A=0x67452301, B=0xefcdab89, C=0x98badcfe, D=0x10325476, and E=0xc3d2e1f0.

The main loop of the algorithm then begins. It processes the message 512 bits at a time and continues for as many 512-bit blocks as are in the message. First the five variables are copied into different variables: a gets A, b gets B, c gets C, d gets D, and e gets E.

The main loop has four rounds of 20 operations each (in contrast to MD5, which has four rounds of 16 operations each). Each operation performs a nonlinear function on three of a, b, c, d, and e, and then does shifting and adding similar to MD5. Shifting the variables accomplishes the same purpose as in MD5 by using different variables in different locations. After all of this, a, b, c, d, and e are added to A, B, C, D, and E respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A, B, C, D, and E.

The SHA is MD5 with the addition of an expanded transformation, and extra round, and better “avalanche” effect. There are no known cryptographic attacks against the SHA-1. Because it produces a 160-bit hash, it is more resistant to brute-force attacks than 128-bit hash functions.

The NIST SHA-1 implementation (using the C language) consists of nine files: Sha.h, Main.c, Config.h, Shautil.obj, Sha.c, Shautil.c, Sha.obj, makefile, and Main.obj. The file Main.c is the driver program for the implementation; it indicates whether to test against FIPS180, to hash a string, or to hash one or more files. The routine Shautil.c merely

computes the SHA of the string “abc” and checks the result against the known hash of “abc”. The file Sha.c is the routine that actually implements the SHA. The implementation is conformant to FIPS180-1, and was tested against reference data in the FIPS, as well as test data from the Handbook of Applied Cryptography. Results verified against all test data.

Conclusion

In this document we have provided information concerning our selection of implementations for the hashing algorithms CRC32, MD4, MD5, and SHA-1, as well as background and information on the hashing algorithms themselves. Various products use one or more of the hashing algorithms MD4, MD5, SHA-1, and CRC32. In summary, moving from CRC32 to MD4 to MD5 to SHA-1, complexity increases, but so does robustness and degree of error-detection. Which algorithm to use (and which implementation of an algorithm) depends on a number of factors, including degree of security required and machine limitations.

Bibliography

(REF1) Webster’s Ninth New Collegiate Dictionary, 1984, Merriam Webster Inc.
(REF2) A Painless Guide to CRC Error Detection Algorithms Index V3.00, September 1996, http://www.repairfaq.org/filipg/LINK/F_crc_v3.html
(REF3) PKZIP Program, http://www.pkzip.com
(REF4) WINZIP Program, http://www.winzip.com
(REF5) NIST FIPS 71, May 1980
(REF6) Snippets Collection, http://www.brokersys.com/snippets/
(REF7) efg’s Computer Lab Mathematics, Cyclic Redundancy Code Calculator, http://www.efg2.com/lab/Mathematics/CRC.htm
(REF8) Schneier, Bruce: Applied Cryptography, Second Edition, 1996, John Wiley & Sons.
(REF9) IETF RFC1320, R.L.Rivest, “The MD4 Message Digest Algorithm, April 1992
(REF10) A.J.Menezes, P. van Oorschot, S.Vanstone, The Handbook of Applied Cryptography, CRC Press, October 1996
(REF11) IETF RFC1321, R.L.Rivest, “The MD5 Message Digest Algorithm, April 1992
(REF12) NIST FIPS 186, Digital Signature Standard, U.S. Department of Commerce, May 1994
(REF13) NIST FIPS 180-1, Secure Hash Standard, U.S. Department of Commerce, April 1995 (supersedes FIPS 180)
(REF14) Borland Corporation, http://www.borland.com
(REF15) Sun Microsystems Corporation, http://www.sun.com
(REF16) Microsoft Systems Journal, http://www.microsoft.com/msj