On the Complexity Analysis of Message Authentication and Cryptographic Protocol Implementations in Low Cost Embedded Systems

Dimitrios A. Karras
Sterea Hellas Institute of Technology, Automation Dept, Psachna, Evoia, 34400, Greece,

dakarras@teiste.gr, dimitrios.karras@gmail.com, dimitrios.karras@ieee.org

Abstract-- This paper presents an experimental analysis and discussion on implementation issues with regards to the performance of message authentication algorithms in embedded applications. Two such algorithms, namely, MD5 and SHA-1 are investigated when implemented into two different platforms:

1. **32-Bit Processors (Intel processor (Pentium III processor at 1000 MHz))**
2. **8-bit microcontrollers (like PHYTEC miniMODUL-537)**

Computational Complexity as well as CPU-processing time results, regarding the performance of these algorithms, both in the 32-bit processors case as well as in the 8-bit processors case are presented. The CPU-processing time results show that such algorithms could be easily incorporated in embedded applications relying on 8-bit microcontrollers and requiring data integrity assurance and data origin authentication. Such applications could involve electronic mail, system monitoring, data mining, biometric security, secure data communications etc. Although NIST has presented similar analyses regarding cryptographic algorithms [1], algorithms aimed on the other hand at data authentication and security protocols implementation have not been investigated in this context. This is precisely the goal and the contribution of this paper.

Index Terms—Message Digests Complexity Analysis, Hash Functions, Message Authentication Complexity Analysis, Embedded Systems.

I. Introduction

This paper deals with message authentication algorithms and more specifically with one way hash functions, which are of primary importance for the security mechanisms of telecommunication systems.

One-way hash functions take as input strings of variable length and produce, according to their mapping algorithm, an output of fixed length, generally smaller than the input string, called digest or hash value. One-way hash functions are used in database and file system management and in security protocols such as authentication, data integrity and digital signature mechanisms in telecommunication applications [2,3,4,5,6,7,8]. Regarding their required characteristics, among others, we point out the easiness to compute the digests and the hardness to compute the message from a given digest and to find another message with the same digest value. However, these characteristics are not sufficient for the one-way hash functions to be applied in cryptographic protocols of communication systems. Therefore, collision resistance is required, which means that it is hard to find two or more random messages to have the same digest [6].

Although Message digests are so important in the security mechanisms of telecommunication systems there is no comparative report in the literature regarding their performance characteristics. More emphasis has been drawn in theoretical tests regarding their quality characteristics. Much less emphasis, however, has been put on practical evaluation schemes regarding these quality characteristics. Such practical evaluation methodologies and comparative reports exist with regards only to cryptographic algorithms [1]. The goal of this paper is to present an experimental report for the performance characteristics, regarding computational and processing time complexity of one way hash functions.

In this section, we discuss some aspects of well-known one-way hash functions. The second section is dedicated to the factors that characterize the computational complexity of one-way functions. In the third section, we describe the experimental methodology for evaluating complexity of the algorithms under consideration. Along with MD5 and SHA in this experimental section we include, for comparison reasons the well known CRC algorithms. Finally, we conclude the paper and outline future work on this subject.

Several one-way hash functions have been proposed and are in the meantime extensively used in security mechanisms, such as MD2, MD4, MD5, SHA and functions that rely on symmetric or asymmetric cryptographic systems. In the following, we shortly underline some relevant parameters of the MD5 and the SHA from the points of view under consideration.

The MD5 algorithm handles messages of arbitrary length and produces digests consisting of 128 bits [9, 10, 11]. Each message, after it has been appended by padding bits, is processed in blocks of 512 bits in length. A buffer of 128 bits holds the intermediate and the final results of the one-way function. Four rounds of processing comprise the algorithm. Each round consists of 16 processing steps. The processing relies on values constructed from the sine function, bit rotation and addition modulo 232. Four primitive logical functions are used, where each one of them is used for one out of the four rounds of processing. The logical functions perform a set of bit-wise logical operations and take three 32 bit words as input and produce a 32 bit word as output.

The Secure Hash Algorithm (SHA, [9, 10, 11, 12]) is a variant of the MD4 algorithm, like the MD5 algorithm. However, the algorithm produces digests of 160 bits in length, although it takes as input messages of arbitrary length as the MD5 does. Each message, similarly to the MD5, after it has been appended by padding bits is processed in blocks of 512 bits in length. A buffer of 160 bits holds the intermediate and the final results of the one-way function. The algorithm consists of 80 processing steps. The processing relies on four additive constants and, like the MD5, on bit rotation (circular left shift) and addition modulo 232. Three different primitive logical functions are used, each one of them for a corresponding step of the processing. Each logical function performs a set of bit-wise logical operations, takes three 32 bit words as input and produces a 32 bit word as output.

II. One way hash function computational and processing time complexity issues

We first, more formally discuss one way hash functions.

A one-way hash function, H(M), operates on an arbitrary-length pre-image message, M. It returns a fixed-length hash value, h.

h = H(M), where h is of length m

Many functions can take an arbitrary-length input and return an output of fixed length, but one-way hash functions have additional characteristics that make them one-way:

Given M, it is easy to compute h.

Given h, it is hard to compute M such that H(M) =h.

Given M, it is hard to find another message, M', such that H(M) = H(M').

The whole point of the one-way hash function is to provide a "fingerprint" of M that is unique. In some applications, one-way ness is insufficient; we need an additional requirement called collision-resistance; It is hard to find two random messages, M and M', such that H(M) = H(M'). Hash functions of 64 bits are just too small to survive a birthday attack. Most practical one-way hash functions produce 128-bit hashes. This forces anyone attempting the birthday attack to hash 264 random documents to find two that hash to the same value, not enough for lasting security. NIST, in its Secure Hash Standard (SHS), uses a 160-bit hash value. This makes the birthday attack even harder, requiring 2^{80} random hashes.

The following method has been proposed to generate a longer hash value than a given hash function produces.

(1) Generate the hash value of a message, using a one-way hash function.

(2) Prepend the hash value to the message.

(3) Generate the hash value of the concatenation of the message and the hash value.

(4) Create a larger hash value consisting of the hash value generated in step (1) concatenated with the hash value generated in step (3).

(5) Repeat steps (1) through (3) as many times as we wish.

It's not easy to design a function that accepts an arbitrary-length input, let alone make it one-way. In the real world, one-way hash functions are built on the idea of a compression function. This one-way function outputs a hash value of length n given an input of some larger length m. The inputs to the compression function are a message block and the output of the previous blocks of text (see Figure 1). The output is the hash of all blocks up to that point. That is, the hash of block Mj is hj = f(Mj,hj - 1)

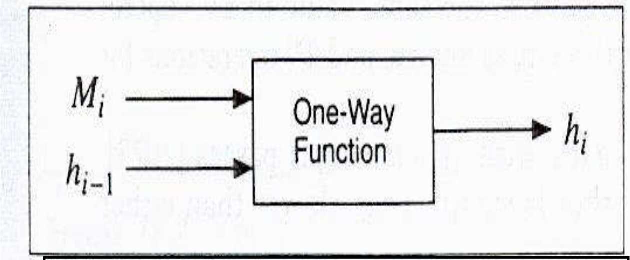


Fig. 1 Block based derivation of one way hash functions

This hash value, along with the next message block, becomes the next input to the compression function. The hash of the entire message is the hash of the last block. The pre-image should contain some kind of binary representation of the length of the entire message. This technique overcomes a potential security problem resulting from messages with different lengths possibly hashing to the same value. This technique is sometimes called MD-strengthening. Various researchers have theorized that if the compression function is secure, then this method of hashing an arbitrary-length pre-image is also secure-but nothing has been proved.

II.1 SECURE HASH STANDARD

NIST [12], along with the NSA, designed the Secure Hash Algorithm (SHA) for use with the Digital Signature Standard. (The standard is the Secure Hash Standard (SHS); SHA is the algorithm used in the standard.) According to the Federal Register:

A Federal Information Processing Standard (FIPS) for Secure Hash Standard (SHS) is being proposed. This proposed standard specified a Secure Hash Algorithm (SHA) for use with the proposed Digital Signature Standard. Additionally, for applications not requiring a digital signature, the SHA is to be used whenever a secure hash algorithm is required for Federal applications. SHA produces a 160-bit hash, longer than MD5. This Standard specifies a Secure Hash Algorithm, SHA-1, for computing a condensed representation of a message or a data file. When a message of any length < 2^{64} bits is input, the SHA-1 produces a 160-bit output called a message digest. The message digest can then be input to the Digital Signature Algorithm (DSA) which generates or verifies the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process because the message digest is usually much smaller in size than the message. The same hash algorithm must be used by the verifier of a digital signature as was used by the creator of the digital signature. 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 very high probability, result in a different message digest, and the signature will fail to verify. SHA-1 is a technical revision of SHA (FIPS 180). A circular left shift operation has been added to the specifications in section 7, line b, page 9 of FIPS 180 and its equivalent in section 8, line c, page 10 of FIPS 180. This revision improves the security provided by this standard.

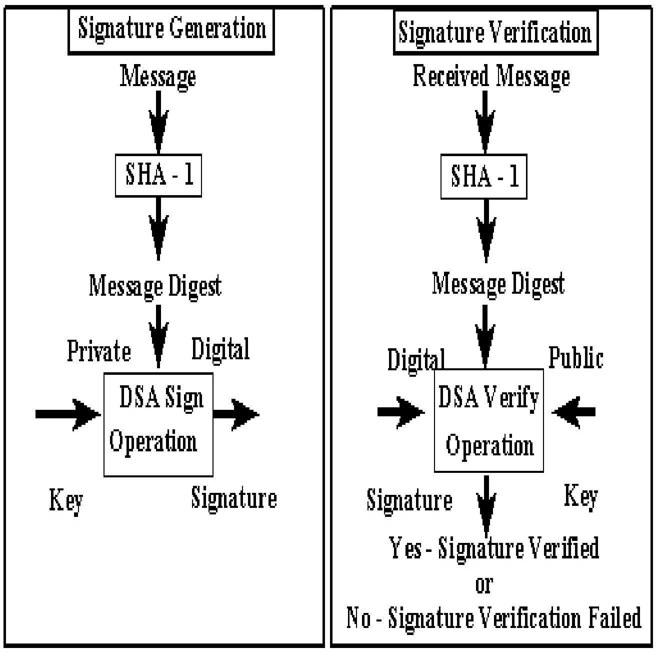


Figure 2: Using the SHA-1 with the DSA

The SHA-1 is designed to have the following properties: 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. The SHA-1 may be used with the DSA in electronic mail, electronic funds transfer, software distribution, data storage, and other applications which require data integrity assurance and data origin authentication. The SHA-1 may also be used whenever it is necessary to generate a condensed version of a message. The SHA-1 may be implemented in software, firmware, hardware, or any combination thereof.

II.2 SECURE HASH STANDARDS CALCULATION ANALYSIS

I) BIT STRINGS AND INTEGER ENCODING FOR SHS CALCULATION

The following terminology related to bit strings and integers will be used:

a. A hex digit is an element of the set {0, 1, ... , 9, A, ... , F}. A hex digit is the representation of a 4-bit string. Examples: 7 = 0111, A = 1010.

b. A word equals a 32-bit string which may be represented as a sequence of 8 hex digits. To convert a word to 8 hex digits each 4-bit string is converted to its hex equivalent as described in (a) above. Example:

1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

c. An integer between 0 and 2^{32} - 1 inclusive may be represented as a word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. Example: the integer 291 = $2^8+2^5+2^1+2^0$ = 256+32+2+1 is represented by the hex word, 00000123.

If z is an integer, $0 <= z < 2^{64}$, then $z = 2^{32}x + y$ where $0 <= x < 2^{32}$ and $0 <= y < 2^{32}$. Since x and y can be represented as words X and Y, respectively, z can be represented as the pair of words (X,Y).

d. block = 512-bit string. A block (e.g., B) may be represented as a sequence of 16 words.

II) WORD OPERATIONS FOR SHS CALCULATION

The following logical operators will be applied to words:
a. Bitwise logical word operations

X ^ Y = bitwise logical "and" of X and Y.

X \/ Y = bitwise logical "inclusive-or" of X and Y.

X XOR Y = bitwise logical "exclusive-or" of X and Y.

~ X = bitwise logical "complement" of X.

Example:
```
      01101100101110011101001001111011
XOR   01100101110000010110100110110111
      --------------------------------
 =    00001001011110001011101111001100
```
b. The operation X + Y is defined as follows: words X and Y represent integers x and y, where $0 <= x < 2^{32}$ and $0 <= y < 2^{32}$. For positive integers n and m, let n mod m be the remainder upon dividing n by m. Compute
$z = (x + y) \bmod 2^{32}$.

Then $0 <= z < 2^{32}$. Convert z to a word, Z, and define Z = X + Y.

c. The circular left shift operation $S^n(X)$, where X is a word and n is an integer with $0 <= n^{32}$, is defined by
$S^n(X) = (X << n) \text{ OR } (X >> 32-n)$.

In the above, X << n is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). X >> n is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus $S^n(X)$ is equivalent to a circular shift of X by n positions to the left.

III) MESSAGE PADDING FOR SHS CALCULATION

The SHA-1 is used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. 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. The following specifies how this padding shall be performed. As a summary, a "1" followed by m "0"s followed by a 64-bit integer are appended to the end of the message to produce a padded message of length 512 * n. The 64-bit integer is L, the length of the original message. The padded message is then processed by the SHA-1 as n 512-bit blocks.
Suppose a message has length $L < 2^{64}$. Before it is input to the SHA-1, the message is padded on the right as follows:

a. "1" is appended. Example: if the original message is "01010000", this is padded to "010100001".

b. "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length L of the original message.
Example: Suppose the original message is the bit string
01100001 01100010 01100011 01100100 01100101.

After step (a) this gives
01100001 01100010 01100011 01100100 01100101 1.
Since L = 40, the number of bits in the above is 41 and 407 "0"s are appended, making the total now 448. This gives (in hex)
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000.

c. Obtain the 2-word representation of L, the number of bits in the original message. If $L < 2^{32}$ then the first word is all zeroes. Append these two words to the padded message.

Example: Suppose the original message is as in (b). Then L= 40 (note that L is computed before any padding). The two-word representation of 40 is hex 00000000 00000028. Hence the final padded message is hex
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028.
The padded message will contain 16 * n words for some n > 0. The padded message is regarded as a sequence of n blocks M_1 , M_2, ... , M_n, where each M_i contains 16 words and M_1 contains the first characters (or bits) of the message.

IV) MESSAGE DIGEST COMPUTATION FOR SHS

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. Five 32-bit variables (MD5 has four variables, but this algorithm needs to produce a 160-bit hash) are initialized as follows:

A = 0x67452301
B = 0xefcdab89
C = 0x98badcfe
D = 0x10325476
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, G gets G, d gets D, and e gets E.
The main loop has four rounds of 20 operations each (MD5 has four rounds of 16 operations each). Each operation performs a nonlinear function on three of a, b, G, d, and e, and then does shifting and adding similar to MD5.
SHA's set of nonlinear functions is:

ft(X,Y,Z) = (XI\ Y) v ((-,X) I\ Z), for t = a to 19.

ft(X,Y,Z) = X +Y + Z, for t = 20 to 39.

ft(X,Y,Z) = (X I\ Y) v (X I\ Z) v (y I\ Z), for t = 40 to 59.

ft(X,Y,Z) =X + Y + Z, for t = 60 to 79.

Four constants are used in the algorithm:

Kt = 0x5a827999, for t = 0 to 19.

Kt = 0x6edgebal, for t = 20 to 39.

Kt = 0x8flbbcdc, for t = 40 to 59.

Kt = 0xca62cld6, for t = 60 to 79.

(those numbers came from: Ox5a827999 = $2^{1/2}/4$, Ox6edgebal = $3^{1/2}/4$, Ox8flbbcdc = $5^{1/2}/4$, and Oxca62cld6 = $10^{1/2}/4$; all times 2^{32}.)
The message block is transformed from 16 32-bit words (M0 to Ml5) to 80 32-bit words (W0 to W79) using the following algorithm:

Wt = Mt, for t = 0 to 15

Wt = (Wt -3 + Wt - 8 + Wt- 14 + Wt- 16) <<< 1, for t = 16 to 79.

(As an interesting aside, the original SHA specification did not have the left circular shift. The change II corrects a technical flaw that made the standard less secure than had been thought. The NSA has refused to elaborate on the exact nature of the flaw.)

If t is the operation number (from 0 to 79), Wt represents the tth sub-block of the expanded message, and <<< s represents a left circular shift of s bits, then the main loop looks like:

```
FOR t = 0 to 79
TEMP = (a <<< 5) + ft(b,c,d) + e + Wt + Kt
e=d
 d=c
c = b <<< 30
b=a
a = TEMP
```

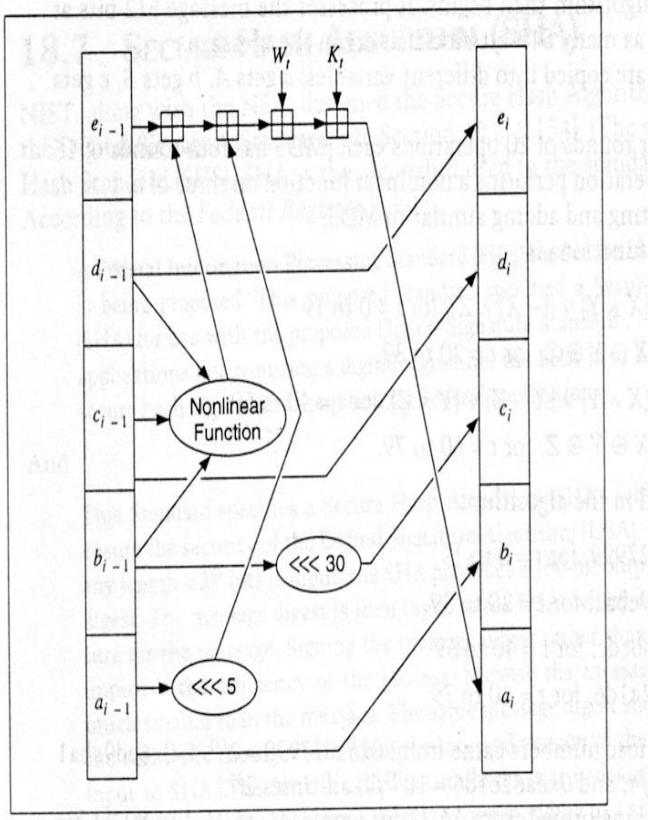


Figure 3. Shifting Operation in SHS

Figure 3 shows one operation. Shifting the variables accomplishes the same thing as MD5 does 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.

SHA is very similar to MD4, but has a 160-bit hash value. The main changes are the addition of an expand transformation and the addition of the previous step's output into the next step for a faster avalanche effect. Ron Rivest made public the design decisions behind MDS, but SHA's designers did not. Below are Rivest's MD5 improvements to MD4 and how they compare with SHA's:
1. " A fourth round has been added." SHA does this, too. However, in SHA the fourth round uses the same f function as the second round.
2. "Each step now has a unique additive constant." SHA keeps the MD4 scheme where it reuses the constants for each group of 20 rounds.
3. "Each step now adds in the result of the previous step. This promotes a faster avalanche effect." This change has been made in SHA as well. The difference in SHA is that a fifth variable is added, and not b, c, or d, which is already used in it. This subtle change makes the den Boer-Bosselaers attack against MD5 impossible against SHA.
4. "The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike." SHA is completely different, since it uses a cyclic error-correcting code.
5. "The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds." SHA uses a constant shift amount in each round. This shift amount is relatively prime to the word size, as in MD4.
This leads to the following comparison: SHA is MD4 with the addition of an expand transformation, an extra round, and better avalanche effect; MD5 is MD4 with improved bit hashing, an extra round, and better avalanche effect.
There are no known cryptographic attacks against SHA. Because it produces a 160-bit hash, it is more resistant to brute-force attacks (including birthday attacks) than 128-bit hash functions.

II.3 THE MD5 HASH FUNCTION

The 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 conjectured that 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 such as RSA. The MD5 algorithm is designed to be quite 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 is "at the edge" in terms of risking successful cryptanalytic attack. 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. First, the message is padded so that its length is just 64 bits short of being a multiple of 512. This padding is a single 1-bit added to the end of the message, followed by as many zeros as are required.

Then, a 64-bit representation of the message's length (before padding bits were added) is appended to the result. These two steps serve to make the message length an exact multiple of 512 bits in length (required for the rest of the algorithm), while ensuring that different messages will not look the same after padding. Four 32-bit variables are initialized:

A = OxO1234567
B = Ox89abcdef
C = Oxfedcba98
D = Ox76543210

These are called chaining variables.

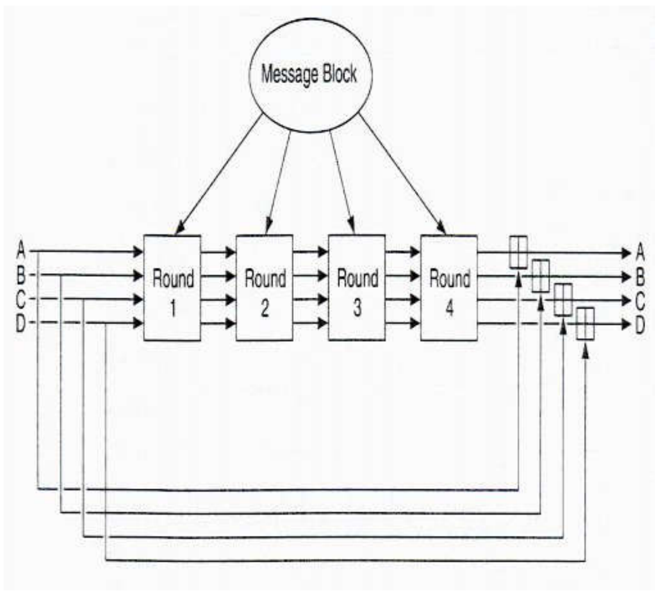


Figure 4. MD5 calculation through 4 rounds

Now, the main loop of the algorithm begins. This loop continues for as many 512bit blocks as are in the message. The four variables are copied into different variables: a gets A, b gets B, c gets C, and d gets D. The main loop has four rounds (MD4 had only three rounds), all very similar. Each round uses a different operation 16 times. Each operation performs a nonlinear function on three of a, b, c, and d.

Then it adds that result to the fourth variable, a subblock of the text and a constant. Then it rotates that result to the right a variable number of bits and adds the result to one of a, b, c; or d. Finally the result replaces one of a, b, c, or d. See Figures 7 and 8.

II.4 MD5 HASH FUNCTION CALCULATION ANALYSIS

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

m_0 m_1 ... m_{b-1}

The following five steps are performed to compute the message digest of the message.

Step 1. Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512. Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

Step 2. Append Length

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. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.) At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M [0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

Step 3. Initialize MD Buffer

A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

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

Step 4. Process Message in 16-Word Blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

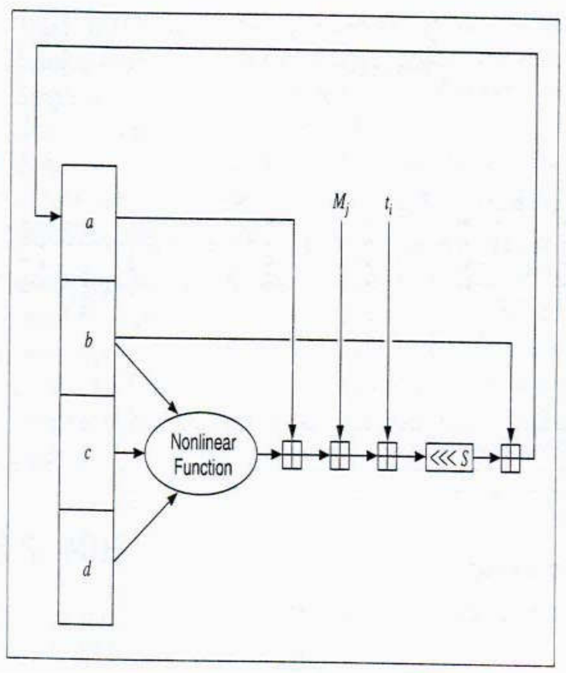
F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))

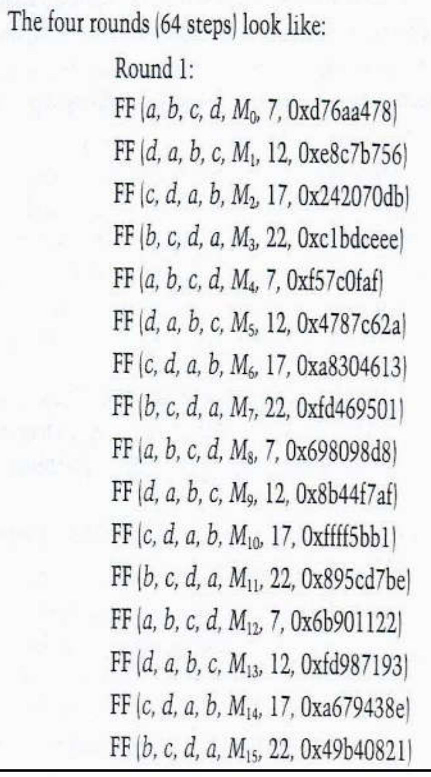
In each bit position F acts as a conditional: if X then Y else Z The function F could have been defined using + instead of v since XY and not (X) Z will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F(X,Y,Z) will be independent and unbiased.
The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of G(X,Y,Z), H(X,Y,Z), and I(X,Y,Z) will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs. This step uses a 64-element table T [1 ... 64] constructed from the sine function. Let T[i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians. We do the following:

```
/* Process each 16-word block. */
  For i = 0 to N/16-1 do

/* Copy block i into X. */
    For j = 0 to 15 do
    Set X[j] to M[i*16+j].
    end /* of loop on j */

/* Save A as AA, B as BB, C as CC, and D as DD. */
    AA = A
    BB = B
    CC = C
    DD = D
```

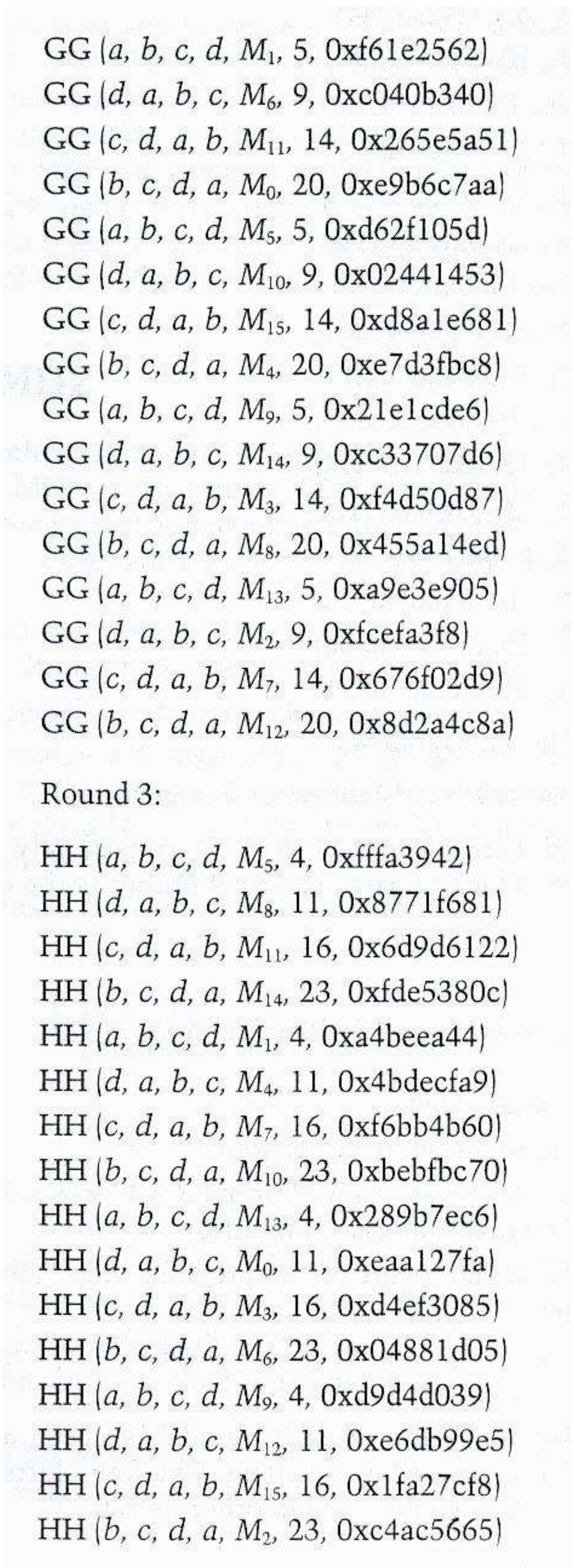



The four rounds (64 steps) look like:

Round 1:

FF (a, b, c, d, M_0, 7, 0xd76aa478)
FF (d, a, b, c, M_1, 12, 0xe8c7b756)
FF (c, d, a, b, M_2, 17, 0x242070db)
FF (b, c, d, a, M_3, 22, 0xc1bdceee)
FF (a, b, c, d, M_4, 7, 0xf57c0faf)
FF (d, a, b, c, M_5, 12, 0x4787c62a)
FF (c, d, a, b, M_6, 17, 0xa8304613)
FF (b, c, d, a, M_7, 22, 0xfd469501)
FF (a, b, c, d, M_8, 7, 0x698098d8)
FF (d, a, b, c, M_9, 12, 0x8b44f7af)
FF (c, d, a, b, M_{10}, 17, 0xffff5bb1)
FF (b, c, d, a, M_{11}, 22, 0x895cd7be)
FF (a, b, c, d, M_{12}, 7, 0x6b901122)
FF (d, a, b, c, M_{13}, 12, 0xfd987193)
FF (c, d, a, b, M_{14}, 17, 0xa679438e)
FF (b, c, d, a, M_{15}, 22, 0x49b40821)

Figure 5. Example of MD5 round 1 outcome



GG (*a*, *b*, *c*, *d*, M_1, 5, 0xf61e2562)
GG (*d*, *a*, *b*, *c*, M_6, 9, 0xc040b340)
GG (*c*, *d*, *a*, *b*, M_{11}, 14, 0x265e5a51)
GG (*b*, *c*, *d*, *a*, M_0, 20, 0xe9b6c7aa)
GG (*a*, *b*, *c*, *d*, M_5, 5, 0xd62f105d)
GG (*d*, *a*, *b*, *c*, M_{10}, 9, 0x02441453)
GG (*c*, *d*, *a*, *b*, M_{15}, 14, 0xd8a1e681)
GG (*b*, *c*, *d*, *a*, M_4, 20, 0xe7d3fbc8)
GG (*a*, *b*, *c*, *d*, M_9, 5, 0x21e1cde6)
GG (*d*, *a*, *b*, *c*, M_{14}, 9, 0xc33707d6)
GG (*c*, *d*, *a*, *b*, M_3, 14, 0xf4d50d87)
GG (*b*, *c*, *d*, *a*, M_8, 20, 0x455a14ed)
GG (*a*, *b*, *c*, *d*, M_{13}, 5, 0xa9e3e905)
GG (*d*, *a*, *b*, *c*, M_2, 9, 0xfcefa3f8)
GG (*c*, *d*, *a*, *b*, M_7, 14, 0x676f02d9)
GG (*b*, *c*, *d*, *a*, M_{12}, 20, 0x8d2a4c8a)

Round 3:

HH (*a*, *b*, *c*, *d*, M_5, 4, 0xfffa3942)
HH (*d*, *a*, *b*, *c*, M_8, 11, 0x8771f681)
HH (*c*, *d*, *a*, *b*, M_{11}, 16, 0x6d9d6122)
HH (*b*, *c*, *d*, *a*, M_{14}, 23, 0xfde5380c)
HH (*a*, *b*, *c*, *d*, M_1, 4, 0xa4beea44)
HH (*d*, *a*, *b*, *c*, M_4, 11, 0x4bdecfa9)
HH (*c*, *d*, *a*, *b*, M_7, 16, 0xf6bb4b60)
HH (*b*, *c*, *d*, *a*, M_{10}, 23, 0xbebfbc70)
HH (*a*, *b*, *c*, *d*, M_{13}, 4, 0x289b7ec6)
HH (*d*, *a*, *b*, *c*, M_0, 11, 0xeaa127fa)
HH (*c*, *d*, *a*, *b*, M_3, 16, 0xd4ef3085)
HH (*b*, *c*, *d*, *a*, M_6, 23, 0x04881d05)
HH (*a*, *b*, *c*, *d*, M_9, 4, 0xd9d4d039)
HH (*d*, *a*, *b*, *c*, M_{12}, 11, 0xe6db99e5)
HH (*c*, *d*, *a*, *b*, M_{15}, 16, 0x1fa27cf8)
HH (*b*, *c*, *d*, *a*, M_2, 23, 0xc4ac5665)

Figure 6. Example of MD5 rounds 2 and 3 outcomes

Those constants, ti, were chosen as follows:

In step i, ti is the integer part of 232*abs(sin(i)), where i is in radians.

After all of this, a, b, c, and d are added to A, B, C, D, respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A, B, C, and D.

The MD5 message-digest algorithm is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length. It is conjectured that 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 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.

The following are the differences between MD4 and MD5:

1. A fourth round has been added.

2. Each step now has a unique additive constant.

3. The function g in round 2 was changed from (XY v XZ v YZ) to (XZ v Y not(Z)) to make g less symmetric.

4. Each step now adds in the result of the previous step. This promotes 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.

1.The shift amounts in each round have been approximately optimized, to yield a faster "avalanche effect." The shifts in different rounds are distinct.

III. One way hash function computational and processing time complexity issues experimental study

In order to evaluate performance of SHA and MD5 especially in embedded applications we have conducted an extensive experimental study comparing performance issues in two different computer platforms. First, in 32-bit Pentium processor based computers and second, in 8-bit microcontrollers, very popular in embedded applications [13].

In the next paragraphs we present the setup and the final results obtained by evaluating these algorithms. In addition, for comparison reasons, we have included a CRC algorithms performance relevant study.

For the SHA-1 algorithm two C source codes were found. One source code is called SHAtest and the other SHAfile.

The two codes were first compiled and executed using Dev-C++ environment for 32-bit processors like Intel (Pentium III at 1000 MHz) processor. Each code consists of three files:

SHAtest:

1. sha1.c
2. sha1.h
3. shatest.c

sha1.c Description:

This file implements the Secure Hashing Standard as defined in FIPS PUB 180-1 published April 17, 1995. The Secure Hashing Standard, which uses the Secure Hashing Algorithm (SHA), produces a 160-bit message digest for a given data stream. In theory, it is highly improbable that two messages will produce the same message digest. Therefore, this algorithm can serve as a means of providing a "fingerprint" for a message.

shatest.c Description:

This file will exercise the SHA1 class and perform the three tests documented in FIPS PUB 180-1.

SHAfile:

1. sha1.c
2. sha1.h
3. sha.c

sha1.c Description:

This file implements the Secure Hashing Standard as defined in FIPS PUB 180-1 published April 17, 1995. The Secure Hashing Standard, which uses the Secure Hashing Algorithm (SHA), produces a 160-bit message digest for a given data stream. In theory, it is highly improbable that two messages will produce the same message digest. Therefore, this algorithm can serve as a means of providing a "fingerprint" for a message. SHA-1 is defined in terms of 32-bit "words". This code was written with the expectation that the processor has at least a 32-bit machine word size. If the machine word size is larger, the code should still function properly. One caveat to that is that the input functions taking characters and character arrays assume that only 8 bits of information are stored in each character. SHA-1 is designed to work with messages less than 2^64 bits long. Although SHA-1 allows a message digest to be generated for messages of any number of bits less than 2^64, this implementation only works with messages with a length that is a multiple of the size of an 8-bit character.

sha.c Description:

This utility will display the message digest (fingerprint) for the specified file(s).

The codes were compiled successfully with no errors and then were executed. For the first code SHAtest, an MS- DOS window was opened after the execution of the code showing to us the message digest of the three input messages:

- TESTA "abc" = message digest (tA9993E36 4706816A BA3E2571 7850C26C 9CD0D89D)
- TESTB "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq" = message digest (t84983E44 1C3BD26E BAAE4AA1 F95129E5 E54670F1)
- TESTC "a" = message digest (t34AA973C D4C4DAA4 F61EEB2B DBAD2731 6534016F)

For the second code SHAfile, the code was executed through MS-Dos and when a file was inserted as an input the message digest of that file was then been calculated.

MD5 C source file for 32-bit Processors

For the MD5 algorithm two C source codes were found. One source code is called MD5code and the other MD5rivest. The two codes were first compiled and executed using Dev-C++ environment for 32-bit processors like Intel (Pentium III at 1000 MHz) processor. Each code consists of four files:

MD5code:

1. global.h
2. md5.h
3. MD5C.c
4. MDDRIVER.c

MD5C.c Description:

This file contains all the transformations and the calculations of the MD5 algorithm.

MDDRIVER.c Description:

This file contains test driver for MD5.

MD5rivest:

1. global.h
2. md5.h
3. MD5C.c
4. MDDRIVER.c

MD5C.c Description:

This file contains all the transformations and the calculations of the MD5 algorithm.

MDDRIVER.c Description:

This file contains test driver for MD2, MD4 and MD5

The codes were compiled successfully with no errors and then were executed. For the first code MD5code, a dos window was opened after the execution of the code showing to us the four possibilities we have:

- First to enter d for demo test
- Second to enter s for a string input message
- Third to enter f for a filename
- Fourth to enter t for time trial

Choosing one of the above lead us to find the MD5 message digest of any input message.

For the second code the MD5rivest code, the code was executed through MS-Dos and when a file was inserted as an input the message digest of that file was then been calculated.

CRC C source file for 32-bit Processors

For the CRC algorithm three C source codes were found. The codes were first compiled and executed using Dev-C++ environment for 32-bit processors like Intel processor. The codes are:

1. crc2.c Computes crc by bit shifting
2. crcfast.c Computes crc by table lookup
3. crctab.c Computes tables used in crcfast.c

The utilities crc2 and crcfast compute the cyclic redundancy checks for both the crc-16 (used in arc files) and crc-ccitt (used in xmodem). crcfast is faster than crc2. These routines compute the crc's for a given file as a means of checking data integrity. All the above routines are short and illustrate the basic principles of crc calculations.
The codes were compiled successfully with no errors and then were executed. An MS-DOS window was opened after the execution of each code showing to us the results.

Machine Word Size and Time functions

One issue that arises in software implementations is the basic underlying architectures. The platforms on which we performed testing were oriented to 32-bit architectures. However, performance on 8-bit machines is also important. It is difficult to project how various architectures will be distributed over the next 30 years. Hence, it is difficult to assign weights to the corresponding performance figures that accurately represent their importance during this timeframe. Nonetheless, the following picture emerges: It appears that over the next 30 years, 8-bit, 32-bit, and 64-bit architectures will all play a significant role (128-bit architectures might be added to the list at some point). Although the 8-bit architectures used in certain applications will gradually be supplanted by 32-bit versions, 8-bit architectures are not likely to disappear. Meanwhile, some 32-bit architectures will be supplanted by 64-bit versions at the high-end, but 32-bit architectures will become increasingly relevant in low-end applications, so that their overall significance will remain high. Meanwhile, 64-bit architectures will grow in importance. Since none of these predictions can be quantified, it appears that versatility is of the essence. Some information on the performance of the three algorithms (SHA-1, MD5, CRC) with respect to word size may be accrued from Tables A.1 and A.3 for 32-bit processors. The tables (A.1-A.3) show an approximation of the processor time since begging of the program, in clock "ticks". The Clock format shown below was used in order to measure the processor time for 32-bit processors:

```
        Clock_t  clock (void)
```

Using the above format to determine the elapsed time the procedure below was added to each of the previous codes to any point of the program for every algorithm (SHA-1, MD5 and CRC) and the time results were then calculated:

```
Clock_t now, later;

Double passed;
……
/* get start clock value */
now=clock();

……..

do some processing

……..
/* get end clock value */
later=clock();
/* compute elapsed seconds*/
passed = (later – now)/(double) CLOCKS_PER_SEC;
printf ( " that took %.10e seconds\n", passed);
```

Using the above procedure to any point of the program gives us the time result we want to measure and as a result we can see how much processor time is needed for every calculation that is created inside the program.

The results obtained using the above described setup are as shown in tables A.1, A.2, A.3 for SHA, MD5 and SRC respectively. The operations calculated in these tables are mentioned in section II.

A.1 SHA calculation complexity –processing time – in 32-Bit Processors (Intel Pentium III at 1000 Mhz)

Blocks	*sha1- process messa-ge block*	*sha1- initia lizati on of first 16 wor-ds*	*sha1 circ ular shif-ts AND oper atio ns*	*sha1 circ ular shift s- XOR oper atio ns*	*sha1 circ ular shif-ts - AND oper atio ns*	*sha1- circu-lar-shifts - XOR ope-ra-tions*	*sha1- Messa ge digest-XOR & AND opera-tions*	*sha1 pad mes-sage blo-ck*
SHA-1 Algo-rithm time	8.8e-001 sec	5.5e-001 sec	9.3e -001 sec	8.8e -001 sec	8.2e -001 sec	8.8e-001 sec	5.0e-001 sec	5.54 sec
Repe-tition cycles	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5

A.2 MD5 calculation complexity –processing time – in 32-Bit Processors (Intel Pentium III at 1000 Mhz)

Blocks	MD5 Block Update operation	MD5 Finali-zation	MD5 basic transfor mation	MD5 Encode input
MD5 Algorithm time	6.0e-001 seconds	9.9e-001 seconds	7.7e-001 seconds	1.1e-001 seconds
Repetition cycle	100000	100000	100000	100000

A.3 CRC calculation complexity –processing time – in 32-Bit Processors (Intel Pentium III at 1000 Mhz)

Blocks	update crc2	Update crc2 reverse	Update crcfast	Update crcfast reverse
CRC Algorith m time	4.0e-001 seconds	5.9e-001 seconds	3.7e-001 seconds	2.2e-001 seconds
Repetitio n cycle	1000000	1000000	1000000	1000000

SHA-1, MD5 and CRC C modified source code for 8-bit microcontrollers (PHYTEC miniMODUL-537)

Using now the Keil Software Chain for programming the 8-bit microcontroller (miniMODUL-537) three hex-files were created. One HEX-file for the SHA-1 algorithm, one HEX-file for the MD5 algorithm and one HEX-file for the CRC algorithm. The previous C source codes that were found for each algorithm were compiled and executed using Dev-C++ for 32-bit processors. Using now the Keil C51 ANSI C compiler the codes were modified in order to match with the needs of the new compiler and the 8-bit microcontroller. Three new codes one for the SHA-1 algorithm, one for the MD5 algorithm and one for the CRC algorithm were created in order to produce the three HEX-files needed for download to the microcontroller external flash memory.

Creation of Hex-file for SHA-1 Algorithm

To create a new project file we selected from the µVision2 menu Project – New Project…. This opens a standard Windows dialog that asks us for the new project file name. We used a separate folder for each project. We can simply use the icon Create New Folder in this dialog to get a new empty folder. Then we selected this folder and entered the file name for the new project, SHAfile. µVision2 created a new project file with the name SHAfile.UV2. Then we added to the project the files of the SHA-1 Keil C source code. The STARTUP.A51 file is the startup code for the most 8051 CPU variants. The startup code clears the data memory and initializes hardware and reentrant stack pointers. This file was also included to the project together with the serinit.h header file. After that we put options for our target hardware the miniMODUL-537 microcontroller and then we translated all source files and line the application with a click on the Build Target toolbar icon. Vision2 creates HEX files with each build process when Create HEX file under Options for Target – Output is enabled. So a hex file for the shafile project was created. FlashTools98 for Windows is a utility program that allows download of user code in *.hex-file format from a host-PC to a PHYTEC Single Board Computer (SBC) via an RS-232 connection. The hex file was downloaded to the microcontroller and the Hyper Terminal was used to see the result which was the calculation of the message digest of the input file.

Creation of Hex-file for MD5 Algorithm

To create a new project file we selected from the µVision2 menu Project – New Project…. This opens a standard Windows dialog that asks us for the new project file name. We used a separate folder for each project. We can simply use the icon Create New Folder in this dialog to get a new empty folder. Then we selected this folder and entered the file name for the new project, MD5. µVision2 created a new project file with the name MD5.UV2. Then we added to the project the files of the MD5 Keil C source code. The STARTUP.A51 file is the startup code for the most 8051 CPU variants. The startup code clears the data memory and initializes hardware and reentrant stack pointers. This file was also included to the project together. After that we putted options for our target hardware the miniMODUL-537 microcontroller and then we translated all source files and line the application with a click on the Build Target toolbar icon. Vision2 creates HEX files with each build process when Create HEX file under Options for Target – Output is enabled. So a hex file for the MD5 project was created. FlashTools98 for Windows is a utility program that allows download of user code in *.hex-file format from a host-PC to a PHYTEC Single Board Computer (SBC) via an RS-232 connection. The hex file was downloaded to the microcontroller and the Hyper Terminal was used to see the result which was the calculation of the message digest of the input file.

Timing results for 8-bit microcontrollers

A.4 SHA-1 calculation complexity –processing time – in 8 Bit microcontrollers

Blocks	*sha1-process messa-ge block*	*sha1-inializ ation of first 16 words*	*sha1-circul ar shifts AND operat ions*	*sha1-circul ar shifts-XOR operat ions*	*sha1-circul ar shifts - AND operat ions*	*sha1-circul ar shifts - XOR operat ions*	*Messa ge digest-XOR and AND opera-tions*	*sha1-pad messa ge block*
sha-1 Algo-rithm CPU Cycles	69500000 cycles	458000000 cycles	344000000 cycles	474000000 cycles	344000000 cycles	344000000 cycles	344000000 cycles	69500000 cycles
Repetition cycles	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5

A.5 MD5 calculation complexity –processing time – in 8 Bit microcontrollers

Blocks	MD5 Block Update operation	MD5 Finali-zation	MD5 basic transfo-rmation	MD5 Encode input
MD5 Algorithm CPU Cycles	80800000 cycles	58700000 cycles	80800000 cycles	34400000 cycles
Repetition cycles	100000	100000	100000	100000

IV. Conclusions

In this paper two very significant message digests algorithms were described in terms of computational /processing time complexity, namely, MD5 and SHA-1. The goal is to understand the requirements of the implementation of such algorithms when they are to be used for embedded applications in secure communication networks. Therefore, the experimental analysis conducted was performed in two computing platforms,

32-Bit Processors (Intel processor (Pentium III processor at 1000 MHz)) and

8-bit microcontrollers (like PHYTEC miniMODUL-537)

The performance of the algorithms to the 32-bit processors was excellent. The speed of calculating the output was very fast. For any input message the message digest of that message was calculated and that message digest was unique for every input. The same performance was also reported to 8-bit microcontrollers. For the SHA-1 algorithm the message digest output was calculated for any multiple of 512-bits because the sha-1 algorithm sequentially processes blocks of 512-bits. The same results were found for the MD5 and CRC algorithms. The timing results from the tables outlined above shows that small times duration were needed for performing any calculation for each algorithm. Finally we can say that these two algorithms can be used in electronic mail, electronic funds transfer, software distribution, data storage, and other applications which require data integrity assurance and data origin authentication even when all these is required to be embedded applications in low cost controllers [13].

References

[1] J. Nechvatal, Report on the Development of the Advanced EWncryption Standard (AES), October 2000, http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf.
[2] I.B. Damgard, ‘Collision Free Hash Functions and Public Key Signature Schemes’, Advances in Cryptography, Eurocrypt ’87, Lecture Notes in Computer Science 304, Springer Verlag, 1987, pp. 203-216.
[3] I.B. Damgard, ‘A Designe Principle for Hash Functions’, Advances in Cryptography, Crypto ’89, Lecture Notes in Computer Science 435, Springer Verlag, 1989, pp. 416-427.
[4] B. Preenel, ‘Cryptographic Hash Functions’, Transactions on Telecommunications, vol. 5, 1994, pp. 431-448.
[5] M.N. Wegman, J.L. Carter, ‘New Hash Functions and Their Use in Authentication and Set Quality’, J. of Computer and System Sciences, vol. 22, 1981, pp. 265-279.
[6] B. Schneier, Applied Cryptography, John Willey and Sons, 1996.
[7] M.Peyravian, A.Roginsky, A.Kshemkalyani, ‘On Probabilities of Hash Value Matches’, J. Computers & Security, Vol. 17, No. 2, 1998, pp. 171-176.
[8] D. Stinson, ‘Combinatorial Techniques for Universal Hashing’, J. of Computer and System Sciences, vol. 48, 1994, pp. 337-346.
[9] W. Stallings, Network and Internetwork Security, Prentice Hall, 1995.
[10] C. P. Pfleeger, Security in Computing, Prentice Hal, 1997.
[11] G. J. Simmons (editor), Contemporary Cryptology, The Science of Information Integrity, IEEE Press, 1992.
[12] FIPS PUB 180-1, Secure Hash Standard, 1995.
[13] The 8051 Microcontroller Architecture, Programming and applications by Kenneth j. Ayala