A Fix of the MD4 Family of Hash Functions - Quasigroup Fold

Danilo Gligoroski[1,2], Smile Markovski[2], Svein Johan Knapskog[1]

1: Centre for Quantifiable Quality of Service in Communication Systems (Q2S)
Centre of Excellence, Norwegian University of Science and Technology (NTNU)
2: Institute of Informatics, Faculty of Natural Sciences, University of Skopje, MACEDONIA

Outline

- [redacted]
- Why quasigroups?
- How – Quasigroup Fold!
- What we gain?
- What we loose?
- Directions ...

Why we propose a fix instead of new hash function?

- Usage of MD5, SHA-1 or some other members of the MD4 family of hash functions can be found in **hundreds of thousands of installed software**:
  - All
  - All
  - All

Simple function call:
md5(data);
or
sha(data);

Why we propose a fix instead of new hash function? (cont.)

- If the fix provides the following features:
  - Makes infeasible all known successful attacks for finding collisions
  - Gives security guarantees according to the current level of mathematical knowledge
  - Doesn't increase the hash output length
  - Doesn't change other "known and good" statistical properties of the hash functions
  - Reduces the costs for intervention in millions of lines of source code of the currently used software
  - As an additional effect it makes some other popular attacks (such as Dictionary attack – unpractical)
  - As a technique can be used generally in the design of new hash functions
- **Then we think that the fix is reasonable option to think about**

Outline

- Why fix?
- [redacted]
- How – Quasigroup Fold!
- What we gain?
- What we loose?
- Directions ...

Why we propose usage of quasigroups?

- All successful collision attacks on MDx family of hash functions has common this:
  - They exploit linear invertability of the operations of
    - Addition modulo 2^{32}
    - Left rotation

  that are operations applied at every step of the hash functions.

Why we propose usage of quasigroups? (cont.)

- den Boer and Bosselaers – ‘91: Collisions on last two rounds of MD4.
  - Solving a system of 32 eq. with 48 unknown 32 bit variables and a system of 16 eq. with 16 unknown 32 bit variables. **Setting up those systems was due to the fact that GG and HH operations of MD4 use linear and invertible addition mod 2^{32} and left rotation.**
- Dobbertin – ’92, … : For MD4 he used weak differential properties of the operation of addition mod 2^{32} in the steps 12-19. He set up 8 eq. with 14 unknown 32 bit variables to find inner almost-collisions. **Setting up those systems was due to the fact that every step operation in MD4 uses linear and invertible addition mod 2^{32} and left rotation.**

Why we propose usage of quasigroups? (cont.)

- den Boer and Bosselaers – ‘93: Collisions on last two rounds of MD5. Usage of clever techniques of choosing “magic numbers” as target values to obtain.
  - Their attack: “walking forward” and “walking backward”, tracing the obtained differences and solving simple linear equations.
  - For example: part 2.2 of their attack explicitly describes equations that are obtained directly from the equations of MD5 that involves **invertible linear operations of addition mod 2^{32} and left rotation.**
- Dobbertin – ‘96: Collisions on full MD5.
  - Crucial role in his attack: Setting up systems of equations for tracing the differences is due to the fact that compression function of MD5 uses **invertible linear operations of addition mod 2^{32} and left rotation.**

Why we propose usage of quasigroups? (cont.)

- Wang at al. 2004, 2005: Collisions on MD4, MD5, HAVAL-128, RIPEMD, SHA-1: Usage of much complicated techniques for tracing the differentials from step to step.
  - Crucial role for setting up all the systems of linear equations that have to be satisfied in order to obtain results with low Hamming distances is the usage of **invertible linear properties of the operations of addition mod 2^{32} and left rotation.**

Why we propose usage of quasigroups? (cont.)

- Quasigroup is such algebraic structure.
-
  - have enormous combinatorial potential

Outline

- Why fix?
- Why quasigroups?
- [redacted]
- What we gain?
- What we loose?
- Directions ...

How is defined the Quasigroup Fold?

- Randomly generated and fixed quasigroup (*Q*,*) of order 16.
- It is:
- Non-commutative,
- Non-associative,
- Non-idempotent
- Without left and right neutral element.

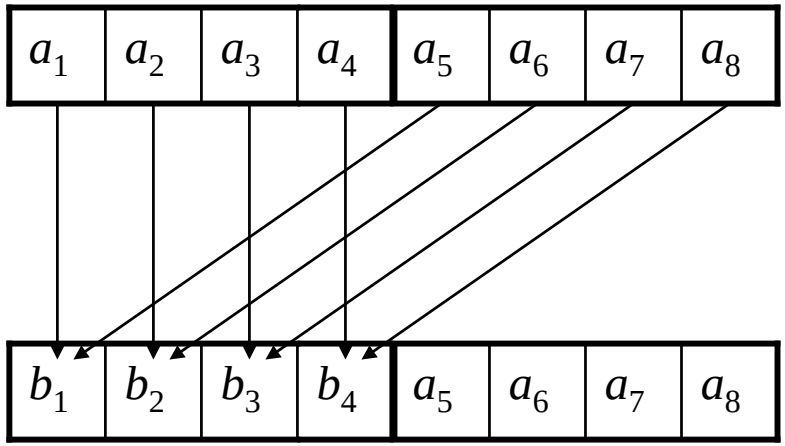
Table 1: A quasigroup $(Q, *)$ of order 16

*	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	a	4	5	9	6	0	e	1	2	c	d	f	3	8	b	7
1	5	b	c	8	4	e	0	7	3	2	f	a	1	9	d	6
2	c	5	2	d	f	8	a	e	1	3	6	7	b	0	9	4
3	7	d	3	e	2	1	b	c	5	9	4	8	0	f	6	a
4	1	2	4	a	b	7	8	9	0	d	3	e	6	c	5	f
5	4	a	8	b	d	2	c	6	e	f	5	9	7	3	1	0
6	0	e	d	2	8	3	6	5	c	b	7	4	9	a	f	1
7	b	6	0	5	9	d	4	8	7	a	2	3	f	1	e	c
8	d	8	6	1	c	a	f	0	b	5	9	2	4	7	3	e
9	2	f	1	0	7	c	5	b	9	6	8	d	a	e	4	3
a	6	c	b	7	a	f	1	3	4	8	e	0	d	5	2	9
b	8	1	f	6	3	9	7	4	a	e	c	5	2	d	0	b
c	f	3	9	4	e	6	2	d	8	7	0	1	c	b	a	5
d	e	9	7	3	1	b	d	f	6	0	a	c	5	4	8	2
e	3	0	e	c	5	4	9	a	f	1	b	6	8	2	7	d
f	9	7	a	f	0	5	3	2	d	4	1	b	e	6	c	8

How is defined the Quasigroup Fold? (cont.)

• Every 32-bit register is seen as a concatenation of 8, 4-bit variables $a_1,\ldots,a_8$.

Quasigroup Fold (applied at the end of every step of the hash function):



$b_1 = a_1 * a_5$

$b_2 = a_6 * a_2$

$b_3 = a_3 * a_7$

$b_4 = a_8 * a_4$

Quasigroup Fold is bijective mapping i.e. it preserves all good statistical properties of MDx.

How is defined the Quasigroup Fold? (cont.)

- Applying current successful attacks on Quasigrouply Folded steps of the hash functions will have to involve two “parastrophes” of the original quasigroup in highly complex and nonlinear way.
- There are no fast algebraic methods for solving nonlinear systems of quasigroup equations in general.

Table 2: The left parastrophe $(Q, \backslash)$ of $(Q, *)$

$\backslash$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	5	7	8	c	1	2	4	f	d	3	0	e	9	a	6	b
1	6	c	9	8	4	0	f	7	3	d	b	1	2	e	5	a
2	d	8	2	9	f	1	a	b	5	e	6	c	0	3	7	4
3	c	5	4	2	a	8	e	0	b	9	f	6	7	1	3	d
4	8	0	1	a	2	e	c	5	6	7	3	4	d	9	b	f
5	f	e	5	d	0	a	7	c	2	b	1	3	6	4	8	9
6	0	f	3	5	b	7	6	a	4	c	d	9	8	2	1	e
7	2	d	a	b	6	3	1	8	7	4	9	0	f	5	e	c
8	7	3	b	e	c	9	2	d	1	a	5	8	4	0	f	6
9	3	2	0	f	e	6	9	4	a	8	c	7	5	b	d	1
a	b	6	e	7	8	d	0	3	9	f	4	2	1	c	a	5
b	e	1	c	4	7	b	3	6	0	5	8	f	a	d	9	2
c	a	b	6	1	3	f	5	9	8	2	e	d	c	7	4	0
d	9	4	f	3	d	c	8	2	e	1	a	5	b	6	0	7
e	1	9	d	0	5	4	b	e	c	6	7	a	3	f	2	8
f	4	a	7	6	9	5	d	1	f	0	2	b	e	8	c	3

Table 3: The right parastrophe $(Q, /)$ of $(Q, *)$

/	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	6	e	7	9	f	0	1	8	4	d	c	a	3	2	b	5
1	4	b	9	8	d	3	a	0	2	e	f	c	1	7	5	6
2	9	4	2	6	3	5	c	f	0	1	7	8	b	e	a	d
3	e	c	3	d	b	6	f	a	1	2	4	7	0	5	8	9
4	5	0	4	c	1	e	7	b	a	f	3	6	8	d	9	2
5	1	2	0	7	e	f	9	6	3	8	5	b	d	a	4	c
6	a	7	8	b	0	c	6	5	d	9	2	e	4	f	3	1
7	3	f	d	a	9	4	b	1	7	c	6	2	5	8	e	0
8	b	8	5	1	6	2	4	7	c	a	9	3	e	0	d	f
9	f	d	c	0	7	b	e	4	9	3	8	5	6	1	2	a
a	0	5	f	4	a	8	2	e	b	7	d	1	9	6	c	3
b	7	1	a	5	4	d	3	9	8	6	e	f	2	c	0	b
c	2	a	1	e	8	9	5	3	6	0	b	d	c	4	f	7
d	8	3	6	2	5	7	d	c	f	4	0	9	a	b	1	e
e	d	6	e	3	c	1	0	2	5	b	a	4	f	9	7	8
f	c	9	b	f	2	a	8	d	e	5	1	0	7	3	6	4

Outline

- Why fix?
- Why quasigroups?
- How – Quasigroup Fold!
- 
- What we loose?
- Directions ...

What we gain?

- Every quasigroup folding introduces 4 nonlinear equations with 8 unknown 4 bit variables.
  - Quasigroup Fold makes impractical all current successful attacks on MD4 family of hash functions.
- Quasigroup Fold Fix on MD4 family of functions (as a software patch) would be much cheaper solution then any “immediate” replacement with new hash function.
- Instead of one fixed one-way hash function (e.g. MD4) with different quasigroups we obtain a family of ~2^{430} one-way hash functions.
  - As a consequence some versions of the dictionary attack can become unpractical
- Quasigroup Fold can be used in the design of any new one-way hash function and will offer a family of one-way functions.

Outline

- Why fix?
- Why quasigroups?
- How – Quasigroup Fold!
- What we gain?
- [redacted]
- Directions ...

What we loose?

- Speed
  - Reference C implementation is 3-5 times slower
  - An optimized assembler version can reduce the loss of the speed to 1.5 – 2 times (our projection)
  - A hardware realization of Quasigroup Fold can be implemented
    - To run in parallel
    - To be executed in 1 or 2 CPU cycles
    - Not to take more then 1000 logic gates
- Compatibility with old MDx functions
  - But we gain a huge family of 2^{430} one-way hash functions with same security characteristics.

Outline

- Why fix?
- Why quasigroups?
- How – Quasigroup Fold!
- What we gain?
- What we loose?
- [redacted]

Directions ...

- Closer look from the scientific community to the security properties of the fixed hash functions
- Fast realization in assembler
- Defining one standardized (default) quasigroup
- Defining protocols that use different randomly generated quasigroups
- Hardware implementation
- Definition of Quasigroup Fold for 64bit CPUs
- Definition of similar register operations with quasigroups

Thank you for your attention

A Fix of the MD4 Family of Hash Functions - Quasigroup Fold

D. Gligoroski* **, S. Markovski** and S.J. Knapskog*

* Centre for Quantifiable Quality of Service in Communication Systems, Norwegian University of Science and Technology, O.S.Bragstads plass 2E, N-7491 Trondheim, NORWAY, e-mail: gligoroski@yahoo.com, Svein.J.Knapskog@Q2S.ntnu.no

** University - "Ss Cyril and Methodius", Faculty of Natural Sciences and Mathematics, Institute of Informatics, P.O.Box 162, 1000 Skopje, Republic of MACEDONIA, e-mail: smile@ii.edu.mk

Abstract

In this paper we show a relatively simple and relatively cheap fix to all hash functions from the MD4 family. Our fix introduces 4 additional quasigroup transformations on 32-bit intermediate variables in every step of the hash function. These operations are called quasigroup-folding. The result is that every equation (assignment) that is defined in the hash function is folded by quasigroup operations that are chosen to be non-commutative, non-associative, non-idempotent, non-involutory and without neutral elements. That is quite different to the Boolean functions conjunction, disjunction, negation, addition ($mod\ 2^{32}$) and exclusive disjunction, that are used in the MD4 family of hash functions. All of these Boolean functions satisfy many algebraic laws suitable for making reductions and for solving equations and systems of equations. On the other hand, solving equations that introduce quasigroup operations can not lead to the reduction of variables, and successful tracing of differences that is the core of every successful differential attack on the hash functions that we know of today. The folding technique involving quasigroup operations can be used also in design of any new hash function. [1]

[1]This work was carried out during the tenure of an ERCIM fellowship of D. Gligoroski visiting Q2S - Centre for Quantifiable Quality of Service in Communication Systems at Norwegian University of Science and Technology - Trondheim, Norway.

1 Introduction

The MD4 family of which the hash functions MD4, MD5, SHA-1, SHA-256, SHA-512, RIPEMD, RIPEMD-160 and many others are members seems to have come to an end of its life. During this 15 years of their existence, for many of them was proved that they can not be considered as one-way hash functions that are collision free. The numerous attacks starting by the attacks of den Boer, Bosselaers and Dobbertin to MD4 and MD5 finally polished to excellence by Wang at al. to SHA-1, have shown that the MD4 family of hash functions no longer defend their status as one-way functions.

All the above mentioned attacks exploit one common characteristic of the MD4 family of hash functions: in every iterative step, the assignment of the new 32-bit working variable is done by applying addition modulo 2^{32} and left rotation. Both of these operations are invertible, and even more, addition modulo 2^{32} is a group operation. Thus, no matter how complicated the unfolding and tracing of all equations involved in the definition of the hash function would seem, in the search for collisions the equations can be unfolded, and then by applying classical algebraic routines for reduction of variables and tracing the differences, the equations can be solved as a linear system of equations in a finite field.

In this paper we show if that common characteristic of MD4 family of hash function is removed, then the existing differential attacks on all hash func-

tions from the MD4 family would be ineffective. To achieve that goal, we choose an algebraic structure that has enormous combinatorial and structural potential, but at the same time is non-commutative, non-associative, non-idempotent, non-involutory and does not have a neutral element (neither left nor right). A quasigroup is such an algebraic structure.

The structure of the paper is the following: In Section 2 we give a brief overview of successful attacks on hash functions from the MD4 family and the crucial techniques and ideas that they employ, in Section 3 we describe the fix for the MD4 family of hash functions (and especially implement it on MD4, MD5 and SHA-1) and discuss why the current attacks would not be successful in that case, and in Section 4 we give conclusions. We have compiled an Appendix 1 where we present a C source code for the modified MD4, MD5 and SHA-1 that have working names: MD4Q, MD5Q and SHA-1Q.

2 A brief summary of successful attacks on hash functions from the MD4 family

In this section we will start by a brief description of the hash functions MD4, MD5 and SHA-1. The description is not detailed, and the reader can find complete description of them in [1, 2, 3]. We will use the notation that Dobbertin used in his paper [4].

The compression function of MD4 uses three Boolean functions

$$\begin{aligned} &F(U,V,W) = (U \wedge V) \vee (\neg U \wedge W),\\ &G(U,V,W) = (U \wedge V) \vee (U \wedge W) \vee (V \wedge W),\\ &H(U,V,W) = U \oplus V \oplus W \end{aligned}$$

that acts bitwise on 32-bit boolean vectors U, V and W. It uses two additive constants and it updates the value of the variable a (which is one of four working 32-bit variables a, b, c and d) by one of the three assignments $FF(a,b,c,d,Z,s)$, $GG(a,b,c,d,Z,s)$ and $HH(a,b,c,d,Z,s)$ that are defined as follows:

$$\begin{aligned} FF &: a := (a + F(b,c,d) + Z)^{\lll s},\\ GG &: a := (a + G(b,c,d) + Z)^{\lll s},\\ HH &: a := (a + H(b,c,d) + Z)^{\lll s}. \end{aligned}$$

MD4 has 3 rounds of 16 steps each, and in each step one of the working 32-bit variables is updated.

The compression function of MD5 uses four Boolean functions

$$\begin{aligned} &F(U,V,W) = (U \wedge V) \vee (\neg U \wedge W),\\ &G(U,V,W) = (U \wedge W) \vee (V \wedge \neg W),\\ &H(U,V,W) = U \oplus V \oplus W,\\ &I(U,V,W) = (U \vee \neg W) \oplus V \end{aligned}$$

It uses 64 additive constants and it updates the value of the variable a (which is one of four working 32-bit variables a, b, c and d) by one of the four assignments $FF(a,b,c,d,Z,Y,s)$, $GG(a,b,c,d,Z,Y,s)$, $HH(a,b,c,d,Z,Y,s)$ and $II(a,b,c,d,Z,Y,s)$ that are defined as follows:

$$\begin{aligned} FF &: a := b + (a + F(b,c,d) + Z + Y)^{\lll s},\\ GG &: a := b + (a + G(b,c,d) + Z + Y)^{\lll s},\\ HH &: a := b + (a + H(b,c,d) + Z + Y)^{\lll s},\\ II &: a := b + (a + I(b,c,d) + Z + Y)^{\lll s}. \end{aligned}$$

MD5 has 4 rounds of 16 steps each, and in each step one of the working 32-bit variables is updated.

Finally, we give brief description of main components in the definition of SHA-1. It uses the same three Boolean functions as MD4, but in 4 rounds and in the following order:

$$\begin{aligned} &F(U,V,W) = (U \wedge V) \vee (\neg U \wedge W),\\ &H(U,V,W) = U \oplus V \oplus W,\\ &G(U,V,W) = (U \wedge V) \vee (U \wedge W) \vee (V \wedge W),\\ &H(U,V,W) = U \oplus V \oplus W \end{aligned}$$

It uses four additive constants and has five working 32-bit variables: a, b, c, d and e. Every round has 20 steps, and it uses an internal procedure for message expansion from 16 to 80, 32-bit variables. Its output is 160 bits. The assignments of working variables is

done by the following functions:

$$t := a^{\lll 5} + F(b,c,d) + e + W + K,$$
$$(a,b,c,d,e) := (t, a, b^{\lll 30}, c, d),$$

$$t := a^{\lll 5} + H(b,c,d) + e + W + K,$$
$$(a,b,c,d,e) := (t, a, b^{\lll 30}, c, d),$$

$$t := a^{\lll 5} + G(b,c,d) + e + W + K,$$
$$(a,b,c,d,e) := (t, a, b^{\lll 30}, c, d),$$

$$t := a^{\lll 5} + H(b,c,d) + e + W + K,$$
$$(a,b,c,d,e) := (t, a, b^{\lll 30}, c, d)$$

where W are 32-bit variables obtained by message expansion and K can have one of four predefined values of additive constants.

The first published successful attack on the last two rounds of MD4 was made by den Boer and Bosselaers in [5]. One of the crucial parts of their attack is solving 32 equations with 48 unknown 32-bit variables, and then again solving a system of 16 equations with 16 unknown 32-bit variables. Setting up and successfully solving such a system of equations is due to the fact that operations GG and HH use easily invertible operations of addition modulo 2^{32} and left rotation.

In his attack on the full MD4, Dobbertin [4] used the same principle as den Boer and Bosselaers, with several crucial observations on weak development of differences (caused by the operations of addition modulo 2^{32}) in steps 12-19. In his paper he sets up 8 equations with 14 unknown 32-bit variables for finding inner almost-collisions. Again, obtaining those 8 equations is very easy having in mind invertibility of addition modulo 2^{32} and left rotation. Further on in the paper all other analysis and computing efforts to find collisions of MD4 are again expressed by systems of equations heavily exploiting the fact that every step in MD4 involves invertible operations of addition modulo 2^{32} and left rotation.

The successful series of attacks on MD5 hash function started early in 1993 with the work of den Boer and Bosselaers in their paper [6]. Again, with clever techniques of choosing "magic" numbers as a target values to obtain, they analyze the first two rounds of MD5 finding collisions by "walking forward" and "walking backward" - i.e. tracing the obtained differences, and solving simple linear equations for some parts of the processed message. The part 2.2 of their algorithm explicitly describes equations that are obtained directly from the equations of MD5 that involves invertible operations of addition modulo 2^{32} and left rotation.

Then in 1996 at the rump session of Eurocrypt '96 by the similar techniques as den Boer and Bosselaers, Dobbertin provided even better results on finding collisions for MD5. Again, the crucial role was on the fact that setting up equations for tracing the differences although complicated, was possible due to the fact that compression function of MD5 uses the invertible operations of addition modulo 2^{32} and left rotation.

Finally in 2004 at the rump session of CRYPTO 2004 (as well as at Cryptology ePrint Archive) Wang at al. presented concrete results of breaking MD5, HAVAL-128 and RIPEMD hash functions [7, 8]. Shortly after that, Wang at al. in February 2005 gave the note about their latest findings about SHA-1 and the possibility to find collisions after 2^{69} SHA-1 hash computations [9]. Although the approach of Wang at al. is much broader and more complicated, we can say that again the basic principle of finding the collisions is exploiting the invertible and linear properties of the functions used in MD4 family of hash functions.

3 A Quasigroup fix for the MD4 family of hash functions

In this section we will describe a technique of bijective transformation of a 32-bit variable called 'Quasigroup Fold'. For that purpose we will use one randomly generated and fixed quasigroup $(Q, *)$ of order 16 that is non-commutative, non-associative, non-idempotent and without (left, right) neutral element. An example of such a quasigroup is given in Table 1.

We will give here only basic definitions for quasigroups and quasigroup operations. The reader can find more detailed explanation in [10] and [11].

Definition 1 *A quasigroup is a groupoid* $(Q, *)$ *sat-*

Table 1: A quasigroup $(Q, *)$ of order 16

$*$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	a	4	5	9	6	0	e	1	2	c	d	f	3	8	b	7
1	5	b	c	8	4	e	0	7	3	2	f	a	1	9	d	6
2	c	5	2	d	f	8	a	e	1	3	6	7	b	0	9	4
3	7	d	3	e	2	1	b	c	5	9	4	8	0	f	6	a
4	1	2	4	a	b	7	8	9	0	d	3	e	6	c	5	f
5	4	a	8	b	d	2	c	6	e	f	5	9	7	3	1	0
6	0	e	d	2	8	3	6	5	c	b	7	4	9	a	f	1
7	b	6	0	5	9	d	4	8	7	a	2	3	f	1	e	c
8	d	8	6	1	c	a	f	0	b	5	9	2	4	7	3	e
9	2	f	1	0	7	c	5	b	9	6	8	d	a	e	4	3
a	6	c	b	7	a	f	1	3	4	8	e	0	d	5	2	9
b	8	1	f	6	3	9	7	4	a	e	c	5	2	d	0	b
c	f	3	9	4	e	6	2	d	8	7	0	1	c	b	a	5
d	e	9	7	3	1	b	d	f	6	0	a	c	5	4	8	2
e	3	0	e	c	5	4	9	a	f	1	b	6	8	2	7	d
f	9	7	a	f	0	5	3	2	d	4	1	b	e	6	c	8

isfying the law

$$(\forall u, v \in Q)(\exists! x, y \in Q)(u * x = v,\ y * u = v).$$

Hence, a quasigroup satisfies the cancellation laws

$$x * y = x * z \implies y = z,\ y * x = z * x \implies y = z$$

and the equations $a * x = b,\ y * a = b$ have unique solutions $x,\ y$ for each $a, b \in Q$. If $(Q, *)$ is a quasigroup, then $*$ is called a quasigroup operation.

Here we consider only finite quasigroups, i.e Q is a finite set. Closely related combinatorial structures to finite quasigroups are the so called Latin squares: a Latin square L on a finite set Q (with cardinality $|Q| = s$) is an $s \times s$-matrix with elements from Q such that each row and each column of the matrix is a permutation of Q. To any finite quasigroup $(Q, *)$ given by its multiplication table there is an associated Latin square L, consisting of the matrix formed by the main body of the table, and each Latin square L on a set Q defines a quasigroup $(Q, *)$.

Given a quasigroup $(Q, *)$ five new operations, so called parastrophes or adjoint operations, can be derived from the operation $*$. We will need only the following two, denoted by $\backslash$ and $/$, and defined by:

$$x * y = z \iff y = x \backslash z \iff x = z/y \quad (1)$$

Then the algebra $(Q, *, \backslash, /)$ satisfies the identities

$$x\backslash(x*y) = y,\ x*(x\backslash y) = y,\ (x*y)/y = x,\ (x/y)*y = x \quad (2)$$

and $(Q, \backslash)$, $(Q, /)$ are quasigroups too. The quasigroup $(Q, \backslash)$ is called left parastrophe and $(Q, /)$ right parastrophe of $(Q, *)$.

The corresponding $(Q, \backslash)$ and $(Q, /)$ of the quasigroup defined in Table 1 are given in Table 2 and 3.

Table 2: The left parastrophe $(Q, \backslash)$ of $(Q, *)$

$\backslash$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	5	7	8	c	1	2	4	f	d	3	0	e	9	a	6	b
1	6	c	9	8	4	0	f	7	3	d	b	1	2	e	5	a
2	d	8	2	9	f	1	a	b	5	e	6	c	0	3	7	4
3	c	5	4	2	a	8	e	0	b	9	f	6	7	1	3	d
4	8	0	1	a	2	e	c	5	6	7	3	4	d	9	b	f
5	f	e	5	d	0	a	7	c	2	b	1	3	6	4	8	9
6	0	f	3	5	b	7	6	a	4	c	d	9	8	2	1	e
7	2	d	a	b	6	3	1	8	7	4	9	0	f	5	e	c
8	7	3	b	e	c	9	2	d	1	a	5	8	4	0	f	6
9	3	2	0	f	e	6	9	4	a	8	c	7	5	b	d	1
a	b	6	e	7	8	d	0	3	9	f	4	2	1	c	a	5
b	e	1	c	4	7	b	3	6	0	5	8	f	a	d	9	2
c	a	b	6	1	3	f	5	9	8	2	e	d	c	7	4	0
d	9	4	f	3	d	c	8	2	e	1	a	5	b	6	0	7
e	1	9	d	0	5	4	b	e	c	6	7	a	3	f	2	8
f	4	a	7	6	9	5	d	1	f	0	2	b	e	8	c	3

Table 3: The right parastrophe $(Q, /)$ of $(Q, *)$

$/$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	6	e	7	9	f	0	1	8	4	d	c	a	3	2	b	5
1	4	b	9	8	d	3	a	0	2	e	f	c	1	7	5	6
2	9	4	2	6	3	5	c	f	0	1	7	8	b	e	a	d
3	e	c	3	d	b	6	f	a	1	2	4	7	0	5	8	9
4	5	0	4	c	1	e	7	b	a	f	3	6	8	d	9	2
5	1	2	0	7	e	f	9	6	3	8	5	b	d	a	4	c
6	a	7	8	b	0	c	6	5	d	9	2	e	4	f	3	1
7	3	f	d	a	9	4	b	1	7	c	6	2	5	8	e	0
8	b	8	5	1	6	2	4	7	c	a	9	3	e	0	d	f
9	f	d	c	0	7	b	e	4	9	3	8	5	6	1	2	a
a	0	5	f	4	a	8	2	e	b	7	d	1	9	6	c	3
b	7	1	a	5	4	d	3	9	8	6	e	f	2	c	0	b
c	2	a	1	e	8	9	5	3	6	0	b	d	c	4	f	7
d	8	3	6	2	5	7	d	c	f	4	0	9	a	b	1	e
e	d	6	e	3	c	1	0	2	5	b	a	4	f	9	7	8
f	c	9	b	f	2	a	8	d	e	5	1	0	7	3	6	4

Let us represent a variable of 32 bits, a as a concatenation of 8, 4-bit variables $a_1, a_2, \ldots, a_8$ i.e. $a = a_1a_2a_3a_4a_5a_6a_7a_8$.

Definition 2 *The operation of quasigroup folding of the variable $a = a_1a_2a_3a_4a_5a_6a_7a_8$ is defined by the following equations:*

$$\begin{array}{rcl} QFOLD(a) & = & a'_1a'_2a'_3a'_4a_5a_6a_7a_8, \\ a'_1 & = & a_1 * a_5, \\ a'_2 & = & a_6 * a_2, \\ a'_3 & = & a_3 * a_7, \\ a'_4 & = & a_8 * a_4 \end{array} \quad (3)$$

From the definition of quasigroup folding operation and from the properties of the quasigroups the following proposition follows.

Proposition 1 *The operation QFOLD as defined by the equation* (3)*, is a bijection on the set $\{0,1\}^{32}$.*

The fix for the MD4 family of hash functions we propose to be the quasigroup folding of variables a in every step of the definition of the hash function.

Thus, the fix for MD4 would look like this:

$$\begin{array}{l} FF: \ a := \mathrm{QFOLD}((a + F(b,c,d) + Z)^{\lll s}), \\ GG: \ a := \mathrm{QFOLD}((a + G(b,c,d) + Z)^{\lll s}), \\ HH: \ a := \mathrm{QFOLD}((a + H(b,c,d) + Z)^{\lll s}). \end{array}$$

The fix for MD5 would be:

$$\begin{array}{l} FF: \ a := \mathrm{QFOLD}(b + (a + F(b,c,d) + Z + Y)^{\lll s}), \\ GG: \ a := \mathrm{QFOLD}(b + (a + G(b,c,d) + Z + Y)^{\lll s}), \\ HH: \ a := \mathrm{QFOLD}(b + (a + H(b,c,d) + Z + Y)^{\lll s}), \\ II: \ a := \mathrm{QFOLD}(b + (a + I(b,c,d) + Z + Y)^{\lll s}). \end{array}$$

and the fix for SHA-1 would be:

$$\begin{array}{l} t := \mathrm{QFOLD}(a^{\lll 5} + F(b,c,d) + e + W + K), \\ (a,b,c,d,e) := (t, a, b^{\lll 30}, c, d), \\ \\ t := \mathrm{QFOLD}(a^{\lll 5} + H(b,c,d) + e + W + K), \\ (a,b,c,d,e) := (t, a, b^{\lll 30}, c, d), \\ \\ t := \mathrm{QFOLD}(a^{\lll 5} + G(b,c,d) + e + W + K), \\ (a,b,c,d,e) := (t, a, b^{\lll 30}, c, d), \\ \\ t := \mathrm{QFOLD}(a^{\lll 5} + H(b,c,d) + e + W + K), \\ (a,b,c,d,e) := (t, a, b^{\lll 30}, c, d) \end{array}$$

Since by Proposition 1, we have that the operation QFOLD is a permutation, it is guaranteed that our fix of MD4 family of hash functions will not introduce some undesirable statistical effects such as shrinking the space of possible outcomes of the hash function or introducing bias of any kind. On the other hand, quasigroup folding will make any algebraic attempt to set up a system of equations that will be easily solved infeasible. The only possible way to trace the equations would be with setting up a system of nonlinear quasigroup equations in a quasigroup of order 16, that is non-commutative, non-associative and without a neutral element.

Let us illustrate that by looking at one of the equations that Dobbertin obtained in his paper [4]. The equation (4) in that paper is:

$$F(\tilde{W}, \tilde{V}, \tilde{U}) - F(W, V, U) = \tilde{Z}^{\lll 13} - Z^{\lll 13}$$

and is obtained by these two equations that are applied in step 15 of MD4:

$$\begin{array}{l} Z := (B + F(W,V,U) + X_{15})^{\lll 19}, \\ \tilde{Z} := (B + F(\tilde{W}, \tilde{V}, \tilde{U}) + X_{15})^{\lll 19}. \end{array}$$

If quasigroup folding is applied, then the equations in that 15-th step will be:

$$\begin{array}{l} Z := \mathrm{QFOLD}((B + F(W,V,U) + X_{15})^{\lll 19}), \\ \tilde{Z} := \mathrm{QFOLD}((B + F(\tilde{W}, \tilde{V}, \tilde{U}) + X_{15})^{\lll 19}). \end{array}$$

Let us denote 32-bit variables Z and $\tilde{Z}$ as 8 concatenated 4-bit variables, i.e. $Z = z_1z_2z_3z_4z_5z_6z_7z_8$ and $\tilde{Z} = \tilde{z}_1\tilde{z}_2\tilde{z}_3\tilde{z}_4\tilde{z}_5\tilde{z}_6\tilde{z}_7\tilde{z}_8$. Then, to obtain the original values of the variables Z and $\tilde{Z}$, first this 8 quasigroup equations would have to be set

$$\begin{array}{l} z_1 = z'_1 / z_5, \\ z_2 = z_6 \setminus z'_2, \\ z_3 = z'_3 / z_7, \\ z_4 = z_8 \setminus z'_4, \\ \\ \tilde{z}_1 = \tilde{z}'_1 / \tilde{z}_5, \\ \tilde{z}_2 = \tilde{z}_6 \setminus \tilde{z}'_2, \\ \tilde{z}_3 = \tilde{z}'_3 / \tilde{z}_7, \\ \tilde{z}_4 = \tilde{z}_8 \setminus \tilde{z}'_4, \end{array}$$

and then the equation

$$F(\tilde{W}, \tilde{V}, \tilde{U}) - F(W, V, U) = \tilde{Z}^{\lll 13} - Z^{\lll 13}$$

would hold. More specifically, from the definition of the quasigroup fold it follows that every referencing of an equation used in some step of the execution of a MD4 hash function introduces at least 4 nonlinear quasigroup equations with 8 unknown variables from the set $\{0, 1, ..., 15\}$. Thus, by applying quasigroup folding in every step further, the final system will end up as a huge nonlinear system of quasigroup equations. Its solution would demand a huge combinatorial effort to try every possible value of the involved variables.

As an additional positive effect that we want to stress from applying the quasigroup folding technique is the enormous number of possibilities for choosing a quasigroup of order 16. Approximately their number can be calculated from [14] and is more then 2^{430}. Thus by applying the quasigroup folding technique on MDx with different and randomly chosen quasigroups of order 16, we are dealing not only with several one-way hash functions but with several huge families of one-way hash functions. In some protocols for network access this fact can be used to eliminate the well known dictionary attack, since attackers in such a case beside the knowledge which particular MDx hash function was used will have to know the quasigroup that is used too, which can be kept secret.

4 Conclusion

We have suggested fixes to the MD4 family of hash functions by means of quasigroup transformations. The introduced quasigroup folding will make any algebraic attempt to set up a system of equations that will be easily solved infeasible. As far as we know, there are no algebraic methods for solving nonlinear systems of quasigroup equations in general (and it is easy to make sure that the quasigroup does not have any algebraic property that makes solving the equations efficiently). Thus, techniques that were have proved to be effective for breaking hash functions from the MD4 family will become ineffective.

As it is always the case, adding a new operation will increase the number of computations. In our proposal we have tried to keep an optimum balance between making the modified MD4 family of hash functions resistant to the current successful attacks and requirements not to decrease the speed of computation of the hash functions too much.

Finally, in accordance to the NIST call for identifying undesirable properties of hash functions, this paper can be seen as a contribution in that direction. The undesirable property of algebraic tractability of the differences in the hash functions can be removed by applying the proposed technique of quasigroup folding in the design of any new hash function. Moreover, the designers of those new one-way hash functions will get not only one particular one-way hash function but a huge family with more then 2^{430} one-way hash functions.

References

[1] Rivest, R.: The MD4 message-digest algorithm, Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.

[2] Rivest, R.: The MD5 message-digest algorithm, Request for Comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, April 1992.

[3] "Secure Hash Standard", United States of American, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180-1, April 1993.

[4] Dobbertin, H.: Cryptanalysis of MD4, J. Cryptology (1998) 11: 253271.

[5] den Boer, B., and Bosselaers, A.: An attack on the last two rounds of MD4, Advances in Cryptology, CRYPTO91, Lecture Notes in Computer Science, vol. 576, Springer-Verlag, Berlin, 1992, pp. 194203.

[6] den Boer, B., and Bosselaers, A.: Collisions for the compression function of MD5, Advances in Cryptology, EUROCRYPT93, Lecture Notes in Computer Science, vol. 765, Springer-Verlag, Berlin, 1994, pp. 293304.

[7] Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, rump session, CRYPTO 2004.

[8] Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive, Report 2004/199, first version (August 16, 2004), second version (August 17, 2004), http://eprint.iacr.org/2004/199.pdf

[9] Wang, X., Lai, X., Yu, H.,: Collision Search Attacks on SHA1, http://theory.csail.mit.edu/ yiqun/shanote.pdf, February 2005.

[10] J. Dénes and A.D. Keedwell, Latin Squares and their Applications, English Univer. Press Ltd. (1974)

[11] Markovski, S., Gligoroski, D., Bakeva, V.: Quasigroup String Processing: Part 1. Contributions, Sec. Math. Tech. Sci., MANU **XX**, 1-2 (1999) 13–28

[12] Markovski, S., Gligoroski, D., Bakeva, V.: Quasigroup and Hash Functions, Disc. Math. and Appl, Sl.Shtrakov and K. Denecke ed., Proceedings of the 6th ICDMA, Bulgaria 2001, 43–50

[13] Gligoroski, D., Markovski, S., Bakeva, V.: On Infinite Class of Strongly Collision Resistant Hash Functions "EDON-F" with Variable Length of Output. Proc. 1-st Inter. Conf. Mathematics and Informatics for industry MII, Greece, Thessaloniki (2003) 302–308

[14] McKay, B.D., and Rogoyski, E.: Latin squares of order ten, Electronic J. Combinatorics, **2** (1995) #N3.

Appendix 1: C sources of fixed MD4, MD5 and SHA-1 hash functions by quasigroup folding

Here we provide three main C sources that can be used with original test drive programs for MD4, MD5 and SHA-1.

Here is the source for MD4Q.C

```
/* MD4CQ.C - An altered MD4 algorithm that fixes the weaknesses of
   MD4 message-digest algorithm
*/

/* The modification made by Danilo Gligoroski 20.06.2005.

   Danilo Gligoroski makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   This work was carried out during the tenure of an ERCIM fellowship
   of D. Gligoroski visiting Q2S - Centre for Quantifiable Quality of
   Service in Communication Systems at Norwegian University of Science
   and Technology - Trondheim, Norway.
*/

/* MD4C.C - RSA Data Security, Inc., MD4 message-digest algorithm
 */

/* Copyright (C) 1990-2, RSA Data Security, Inc. All rights
reserved.

   License to copy and use this software is granted provided that it
   is identified as the "RSA Data Security, Inc. MD4 Message-Digest
   Algorithm" in all material mentioning or referencing this software
   or this function.

   License is also granted to make and use derivative works provided
   that such works are identified as "derived from the RSA Data
   Security, Inc. MD4 Message-Digest Algorithm" in all material
   mentioning or referencing the derived work.

   RSA Data Security, Inc. makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   These notices must be retained in any copies of any part of this
   documentation and/or software.
 */

#include "global.h" #include "md4.h"

/* This is the definition of the quasigroup Q of order 16x16. */
unsigned char Q[256] ={ 10,  4,  5,  9,  6,  0, 14,  1,  2, 12,
13, 15,  3,  8, 11,  7,
 5, 11, 12,  8,  4, 14,  0,  7,  3,  2, 15, 10,  1,  9, 13,  6,
12,  5,  2, 13, 15,  8, 10, 14,  1,  3,  6,  7, 11,  0,  9,  4,
 7, 13,  3, 14,  2,  1, 11, 12,  5,  9,  4,  8,  0, 15,  6, 10,
 1,  2,  4, 10, 11,  7,  8,  9,  0, 13,  3, 14,  6, 12,  5, 15,
 4, 10,  8, 11, 13,  2, 12,  6, 14, 15,  5,  9,  7,  3,  1,  0,
 0, 14, 13,  2,  8,  3,  6,  5, 12, 11,  7,  4,  9, 10, 15,  1,
11,  6,  0,  5,  9, 13,  4,  8,  7, 10,  2,  3, 15,  1, 14, 12,
13,  8,  6,  1, 12, 10, 15,  0, 11,  5,  9,  2,  4,  7,  3, 14,
 2, 15,  1,  0,  7, 12,  5, 11,  9,  6,  8, 13, 10, 14,  4,  3,
 6, 12, 11,  7, 10, 15,  1,  3,  4,  8, 14,  0, 13,  5,  2,  9,
 8,  1, 15,  6,  3,  9,  7,  4, 10, 14, 12,  5,  2, 13,  0, 11,
15,  3,  9,  4, 14,  6,  2, 13,  8,  7,  0,  1, 12, 11, 10,  5,
14,  9,  7,  3,  1, 11, 13, 15,  6,  0, 10, 12,  5,  4,  8,  2,
 3,  0, 14, 12,  5,  4,  9, 10, 15,  1, 11,  6,  8,  2,  7, 13,
 9,  7, 10, 15,  0,  5,  3,  2, 13,  4,  1, 11, 14,  6, 12,  8
};

/* First 16 bits of a variable 'a' will be changed by 16 bits
obtained by quasigroup transformation defined below. */ #define
QFOLD(a) {\
    (a) = ((a)&0x0fffffff)|(((unsigned char)Q[( ((a)&0xf0000000)>>28)| \
    ( ((a)&0x0000f000)>> 8)])<<28);\
    (a) = ((a)&0xf0ffffff)|(((unsigned char)Q[( ((a)&0x0f000000)>>20)| \
    ( ((a)&0x00000f00)>> 8)])<<24);\
    (a) = ((a)&0xff0fffff)|(((unsigned char)Q[( ((a)&0x00f00000)>>20)| \
    ( ((a)&0x000000f0)    )])<<20);\
    (a) = ((a)&0xfff0ffff)|(((unsigned char)Q[( ((a)&0x000f0000)>>12)| \
    ( ((a)&0x0000000f)    )])<<16);\
}

/* Constants for MD4Transform routine.
```

```
 */
#define S11 3 #define S12 7 #define S13 11 #define S14 19 #define
S21 3 #define S22 5 #define S23 9 #define S24 13 #define S31 3
#define S32 9 #define S33 11 #define S34 15

static void MD4Transform PROTO_LIST ((UINT4 [4], unsigned char
[64])); static void Encode PROTO_LIST
  ((unsigned char *, UINT4 *, unsigned int));
static void Decode PROTO_LIST
  ((UINT4 *, unsigned char *, unsigned int));
static void MD4_memcpy PROTO_LIST ((POINTER, POINTER, unsigned
int)); static void MD4_memset PROTO_LIST ((POINTER, int, unsigned
int));

static unsigned char PADDING[64] = {
  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G and H are basic MD4 functions.
 */
#define F(x, y, z) (((x) & (y)) | ((~x) & (z))) #define G(x, y, z)
(((x) & (y)) | ((x) & (z)) | ((y) & (z))) #define H(x, y, z) ((x)
^ (y) ^ (z))

/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG and HH are transformations for rounds 1, 2 and 3 */ /*
Rotation is separate from addition to prevent recomputation */

#define FF(a, b, c, d, x, s) { \
    (a) += F ((b), (c), (d)) + (x); \
    (a) = ROTATE_LEFT ((a), (s)); \
    QFOLD((a));\
  }
#define GG(a, b, c, d, x, s) { \
    (a) += G ((b), (c), (d)) + (x) + (UINT4)0x5a827999; \
    (a) = ROTATE_LEFT ((a), (s)); \
    QFOLD((a));\
  }
#define HH(a, b, c, d, x, s) { \
    (a) += H ((b), (c), (d)) + (x) + (UINT4)0x6ed9eba1; \
    (a) = ROTATE_LEFT ((a), (s)); \
    QFOLD((a));\
  }

/* MD4 initialization. Begins an MD4 operation, writing a new
context.
 */
void MD4Init (context) MD4_CTX *context;
/* context */ {
  context->count[0] = context->count[1] = 0;

  /* Load magic initialization constants.
   */
  context->state[0] = 0x67452301;
  context->state[1] = 0xefcdab89;
  context->state[2] = 0x98badcfe;
  context->state[3] = 0x10325476;
}

/* MD4 block update operation. Continues an MD4 message-digest
     operation, processing another message block, and updating the
     context.
 */
void MD4Update (context, input, inputLen) MD4_CTX *context;
/* context */ unsigned char *input;
/* input block */ unsigned int inputLen;                     /*
length of input block */ {
  unsigned int i, index, partLen;

  /* Compute number of bytes mod 64 */
  index = (unsigned int)((context->count[0] >> 3) & 0x3F);
  /* Update number of bits */
  if ((context->count[0] += ((UINT4)inputLen << 3))
      < ((UINT4)inputLen << 3))
    context->count[1]++;
  context->count[1] += ((UINT4)inputLen >> 29);

  partLen = 64 - index;

  /* Transform as many times as possible.
   */
  if (inputLen >= partLen) {
    MD4_memcpy
      ((POINTER)&context->buffer[index], (POINTER)input, partLen);
    MD4Transform (context->state, context->buffer);
    for (i = partLen; i + 63 < inputLen; i += 64)
      MD4Transform (context->state, &input[i]);

    index = 0;
  }
  else
    i = 0;

  /* Buffer remaining input */
  MD4_memcpy
    ((POINTER)&context->buffer[index], (POINTER)&input[i],
     inputLen-i);
}

/* MD4 finalization. Ends an MD4 message-digest operation, writing
the
     the message digest and zeroizing the context.
 */
void MD4Final (digest, context) unsigned char digest[16];
/* message digest */ MD4_CTX *context;
/* context */ {
  unsigned char bits[8];
  unsigned int index, padLen;

  /* Save number of bits */
  Encode (bits, context->count, 8);

  /* Pad out to 56 mod 64.
   */
  index = (unsigned int)((context->count[0] >> 3) & 0x3f);
  padLen = (index < 56) ? (56 - index) : (120 - index);
  MD4Update (context, PADDING, padLen);

  /* Append length (before padding) */
  MD4Update (context, bits, 8);
  /* Store state in digest */
  Encode (digest, context->state, 16);

  /* Zeroize sensitive information.
   */
  MD4_memset ((POINTER)context, 0, sizeof (*context));
}

/* MD4 basic transformation. Transforms state based on block.
 */
static void MD4Transform (state, block) UINT4 state[4]; unsigned
char block[64]; {
  UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

  Decode (x, block, 64);

  /* Round 1 */
  FF (a, b, c, d, x[ 0], S11); /* 1 */
  FF (d, a, b, c, x[ 1], S12); /* 2 */
  FF (c, d, a, b, x[ 2], S13); /* 3 */
  FF (b, c, d, a, x[ 3], S14); /* 4 */
  FF (a, b, c, d, x[ 4], S11); /* 5 */
  FF (d, a, b, c, x[ 5], S12); /* 6 */
  FF (c, d, a, b, x[ 6], S13); /* 7 */
  FF (b, c, d, a, x[ 7], S14); /* 8 */
  FF (a, b, c, d, x[ 8], S11); /* 9 */
  FF (d, a, b, c, x[ 9], S12); /* 10 */
  FF (c, d, a, b, x[10], S13); /* 11 */
  FF (b, c, d, a, x[11], S14); /* 12 */
  FF (a, b, c, d, x[12], S11); /* 13 */
  FF (d, a, b, c, x[13], S12); /* 14 */
  FF (c, d, a, b, x[14], S13); /* 15 */
  FF (b, c, d, a, x[15], S14); /* 16 */

  /* Round 2 */
  GG (a, b, c, d, x[ 0], S21); /* 17 */
  GG (d, a, b, c, x[ 4], S22); /* 18 */
  GG (c, d, a, b, x[ 8], S23); /* 19 */
  GG (b, c, d, a, x[12], S24); /* 20 */
  GG (a, b, c, d, x[ 1], S21); /* 21 */
  GG (d, a, b, c, x[ 5], S22); /* 22 */
  GG (c, d, a, b, x[ 9], S23); /* 23 */
  GG (b, c, d, a, x[13], S24); /* 24 */
  GG (a, b, c, d, x[ 2], S21); /* 25 */
  GG (d, a, b, c, x[ 6], S22); /* 26 */
  GG (c, d, a, b, x[10], S23); /* 27 */
  GG (b, c, d, a, x[14], S24); /* 28 */
  GG (a, b, c, d, x[ 3], S21); /* 29 */
  GG (d, a, b, c, x[ 7], S22); /* 30 */
  GG (c, d, a, b, x[11], S23); /* 31 */
  GG (b, c, d, a, x[15], S24); /* 32 */

  /* Round 3 */
  HH (a, b, c, d, x[ 0], S31); /* 33 */
  HH (d, a, b, c, x[ 8], S32); /* 34 */
  HH (c, d, a, b, x[ 4], S33); /* 35 */
```

```
  HH (b, c, d, a, x[12], S34); /* 36 */
  HH (a, b, c, d, x[ 2], S31); /* 37 */
  HH (d, a, b, c, x[10], S32); /* 38 */
  HH (c, d, a, b, x[ 6], S33); /* 39 */
  HH (b, c, d, a, x[14], S34); /* 40 */
  HH (a, b, c, d, x[ 1], S31); /* 41 */
  HH (d, a, b, c, x[ 9], S32); /* 42 */
  HH (c, d, a, b, x[ 5], S33); /* 43 */
  HH (b, c, d, a, x[13], S34); /* 44 */
  HH (a, b, c, d, x[ 3], S31); /* 45 */
  HH (d, a, b, c, x[11], S32); /* 46 */
  HH (c, d, a, b, x[ 7], S33); /* 47 */
  HH (b, c, d, a, x[15], S34); /* 48 */

  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;

  /* Zeroize sensitive information.
   */
  MD4_memset ((POINTER)x, 0, sizeof (x));
}

/* Encodes input (UINT4) into output (unsigned char). Assumes len
is
     a multiple of 4.
 */
static void Encode (output, input, len) unsigned char *output;
UINT4 *input; unsigned int len; {
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4) {
    output[j] = (unsigned char)(input[i] & 0xff);
    output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
    output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
    output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
  }
}

/* Decodes input (unsigned char) into output (UINT4). Assumes len
is
     a multiple of 4.
 */
static void Decode (output, input, len)

UINT4 *output; unsigned char *input; unsigned int len; {
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4)
    output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
      (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
}

/* Note: Replace "for loop" with standard memcpy if possible.
 */
static void MD4_memcpy (output, input, len) POINTER output;
POINTER input; unsigned int len; {
  unsigned int i;

  for (i = 0; i < len; i++)
    output[i] = input[i];
}

/* Note: Replace "for loop" with standard memset if possible.
 */
static void MD4_memset (output, value, len) POINTER output; int
value; unsigned int len; {
  unsigned int i;

  for (i = 0; i < len; i++)
    ((char *)output)[i] = (char)value;
}
```

Here is the source for MD5Q.C

```
/* MD5CQ.C - An altered MD5 algorithm that fixes the weaknesses of
   MD5 message-digest algorithm
*/

/* The modification made by Danilo Gligoroski 20.06.2005.

   Danilo Gligoroski makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   This work was carried out during the tenure of an ERCIM fellowship
   of D. Gligoroski visiting Q2S - Centre for Quantifiable Quality of
   Service in Communication Systems at Norwegian University of Science
   and Technology - Trondheim, Norway.
*/

/* MD5C.C - RSA Data Security, Inc., MD5 message-digest algorithm
 */

/* Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
 */

#include "global.h" #include "md5.h"

#include <memory.h>

/* This is the definition of the quasigroup Q of order 16x16. */
unsigned char Q[256] ={ 10,  4,  5,  9,  6,  0, 14,  1,  2, 12,
13, 15,  3,  8, 11,  7,
 5, 11, 12,  8,  4, 14,  0,  7,  3,  2, 15, 10,  1,  9, 13,  6,
12,  5,  2, 13, 15,  8, 10, 14,  1,  3,  6,  7, 11,  0,  9,  4,
 7, 13,  3, 14,  2,  1, 11, 12,  5,  9,  4,  8,  0, 15,  6, 10,
 1,  2,  4, 10, 11,  7,  8,  9,  0, 13,  3, 14,  6, 12,  5, 15,
 4, 10,  8, 11, 13,  2, 12,  6, 14, 15,  5,  9,  7,  3,  1,  0,
 0, 14, 13,  2,  8,  3,  6,  5, 12, 11,  7,  4,  9, 10, 15,  1,
11,  6,  0,  5,  9, 13,  4,  8,  7, 10,  2,  3, 15,  1, 14, 12,
13,  8,  6,  1, 12, 10, 15,  0, 11,  5,  9,  2,  4,  7,  3, 14,
 2, 15,  1,  0,  7, 12,  5, 11,  9,  6,  8, 13, 10, 14,  4,  3,
 6, 12, 11,  7, 10, 15,  1,  3,  4,  8, 14,  0, 13,  5,  2,  9,
 8,  1, 15,  6,  3,  9,  7,  4, 10, 14, 12,  5,  2, 13,  0, 11,
15,  3,  9,  4, 14,  6,  2, 13,  8,  7,  0,  1, 12, 11, 10,  5,
14,  9,  7,  3,  1, 11, 13, 15,  6,  0, 10, 12,  5,  4,  8,  2,
 3,  0, 14, 12,  5,  4,  9, 10, 15,  1, 11,  6,  8,  2,  7, 13,
 9,  7, 10, 15,  0,  5,  3,  2, 13,  4,  1, 11, 14,  6, 12,  8
};

/* First 16 bits of a variable 'a' will be changed by 16 bits
obtained by quasigroup transformation defined below. */ #define
QFOLD(a) {\
    (a) = ((a)&0x0fffffff)|(((unsigned char)Q[( ((a)&0xf0000000)>>28)| \
    ( ((a)&0x0000f000)>> 8)])<<28);\
    (a) = ((a)&0xf0ffffff)|(((unsigned char)Q[( ((a)&0x0f000000)>>20)| \
    ( ((a)&0x00000f00)>> 8)])<<24);\
    (a) = ((a)&0xff0fffff)|(((unsigned char)Q[( ((a)&0x00f00000)>>20)| \
    ( ((a)&0x000000f0)    )])<<20);\
    (a) = ((a)&0xffff0fff)|(((unsigned char)Q[( ((a)&0x000f0000)>>12)| \
    ( ((a)&0x0000000f)    )])<<16);\
}

 /* Constants for MD5Transform routine.
 */
#define S11 7 #define S12 12 #define S13 17 #define S14 22 #define
S21 5 #define S22 9 #define S23 14 #define S24 20 #define S31 4
#define S32 11 #define S33 16 #define S34 23 #define S41 6 #define
S42 10 #define S43 15 #define S44 21

static void MD5Transform PROTO_LIST ((UINT4 [4], unsigned char
[64])); static void Encode PROTO_LIST
  ((unsigned char *, UINT4 *, unsigned int));
static void Decode PROTO_LIST
  ((UINT4 *, unsigned char *, unsigned int));
static void MD5_memcpy PROTO_LIST ((POINTER, POINTER, unsigned
int)); static void MD5_memset PROTO_LIST ((POINTER, int, unsigned
int));

static unsigned char PADDING[64] = {
  0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

/* F, G, H and I are basic MD5 functions.
 */
```

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z))) #define G(x, y, z)
(((x) & (z)) | ((y) & (~z))) #define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))

/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
Rotation is separate from addition to prevent recomputation.
 */
#define FF(a, b, c, d, x, s, ac) { \
 (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
 QFOLD((a));\
  }
#define GG(a, b, c, d, x, s, ac) { \
 (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
 QFOLD((a));\
  }
#define HH(a, b, c, d, x, s, ac) { \
 (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
 QFOLD((a));\
  }
#define II(a, b, c, d, x, s, ac) { \
 (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
 (a) = ROTATE_LEFT ((a), (s)); \
 (a) += (b); \
 QFOLD((a));\
  }

/* MD5 initialization. Begins an MD5 operation, writing a new
context.
 */
void MD5Init (context) MD5_CTX *context;
/* context */ {
  context->count[0] = context->count[1] = 0;
  /* Load magic initialization constants.
*/
  context->state[0] = 0x67452301;
  context->state[1] = 0xefcdab89;
  context->state[2] = 0x98badcfe;
  context->state[3] = 0x10325476;
}

/* MD5 block update operation. Continues an MD5 message-digest
  operation, processing another message block, and updating the
  context.
 */
void MD5Update (context, input, inputLen) MD5_CTX *context;
/* context */ unsigned char *input;
/* input block */ unsigned int inputLen;                      /*
length of input block */ {
  unsigned int i, index, partLen;

  /* Compute number of bytes mod 64 */
  index = (unsigned int)((context->count[0] >> 3) & 0x3F);

  /* Update number of bits */
  if ((context->count[0] += ((UINT4)inputLen << 3))
   < ((UINT4)inputLen << 3))
 context->count[1]++;
  context->count[1] += ((UINT4)inputLen >> 29);

  partLen = 64 - index;

  /* Transform as many times as possible.
*/
  if (inputLen >= partLen) {
 MD5_memcpy
   ((POINTER)&context->buffer[index], (POINTER)input, partLen);
 MD5Transform (context->state, context->buffer);

 for (i = partLen; i + 63 < inputLen; i += 64)
   MD5Transform (context->state, &input[i]);

 index = 0;
  }
  else
 i = 0;

  /* Buffer remaining input */
  MD5_memcpy
 ((POINTER)&context->buffer[index], (POINTER)&input[i],
  inputLen-i);
}

/* MD5 finalization. Ends an MD5 message-digest operation, writing
the
  the message digest and zeroizing the context.
 */
void MD5Final (digest, context) unsigned char digest[16];
/* message digest */ MD5_CTX *context;
/* context */ {
  unsigned char bits[8];
  unsigned int index, padLen;

  /* Save number of bits */
  Encode (bits, context->count, 8);

  /* Pad out to 56 mod 64.
*/
  index = (unsigned int)((context->count[0] >> 3) & 0x3f);
  padLen = (index < 56) ? (56 - index) : (120 - index);
  MD5Update (context, PADDING, padLen);

  /* Append length (before padding) */
  MD5Update (context, bits, 8);
  /* Store state in digest */
  Encode (digest, context->state, 16);

  /* Zeroize sensitive information.
*/
  MD5_memset ((POINTER)context, 0, sizeof (*context));
}

/* MD5 basic transformation. Transforms state based on block.
 */
static void MD5Transform (state, block) UINT4 state[4]; unsigned
char block[64]; {
  UINT4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];

  Decode (x, block, 64);

  /* Round 1 */
  FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
  FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
  FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
  FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
  FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
  FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
  FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */
  FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */
  FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */
  FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */
  FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
  FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
  FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
  FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
  FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
  FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

 /* Round 2 */
  GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */
  GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */
  GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
  GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */
  GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */
  GG (d, a, b, c, x[10], S22,  0x2441453); /* 22 */
  GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
  GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */
  GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */
  GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
  GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
  GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */
  GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
  GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */
  GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */
  GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

  /* Round 3 */
  HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */
  HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */
  HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
  HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
  HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */
  HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */
  HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */
  HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
  HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
  HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */
  HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */
  HH (b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */
  HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */
  HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
  HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
  HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */
```

```
  /* Round 4 */
  II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */
  II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */
  II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
  II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */
  II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
  II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */
  II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
  II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */
  II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */
  II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
  II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */
  II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
  II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */
  II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
  II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */
  II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */

  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;

  /* Zeroize sensitive information.
*/
  MD5_memset ((POINTER)x, 0, sizeof (x));
}

/* Encodes input (UINT4) into output (unsigned char). Assumes len
is
  a multiple of 4.
 */
static void Encode (output, input, len) unsigned char *output;
UINT4 *input; unsigned int len; {
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4) {
 output[j] = (unsigned char)(input[i] & 0xff);
 output[j+1] = (unsigned char)((input[i] >> 8) & 0xff);
 output[j+2] = (unsigned char)((input[i] >> 16) & 0xff);
 output[j+3] = (unsigned char)((input[i] >> 24) & 0xff);
  }
}

/* Decodes input (unsigned char) into output (UINT4). Assumes len
is
  a multiple of 4.
 */
static void Decode (output, input, len) UINT4 *output; unsigned
char *input; unsigned int len; {
  unsigned int i, j;

  for (i = 0, j = 0; j < len; i++, j += 4)
 output[i] = ((UINT4)input[j]) | (((UINT4)input[j+1]) << 8) |
   (((UINT4)input[j+2]) << 16) | (((UINT4)input[j+3]) << 24);
}

/* Note: Replace "for loop" with standard memcpy if possible.
 */

static void MD5_memcpy (output, input, len) POINTER output;
POINTER input; unsigned int len; { /*
  unsigned int i;

  for (i = 0; i < len; i++)
 output[i] = input[i];
*/
  memcpy(output,input,len);
}

/* Note: Replace "for loop" with standard memset if possible.
 */
static void MD5_memset (output, value, len) POINTER output; int
value; unsigned int len; { /*
  unsigned int i;

  for (i = 0; i < len; i++)
 ((char *)output)[i] = (char)value;
*/
  memset(output,value,len);
}
```

Here is the source for SHA1Q.C

```
/* SHA1Q.C - An altered SHA-1 algorithm that fixes the weaknesses
of
   SHA-1 message-digest algorithm
*/

/* The modification made by Danilo Gligoroski 20.06.2005.

   Danilo Gligoroski makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   This work was carried out during the tenure of an ERCIM fellowship
   of D. Gligoroski visiting Q2S - Centre for Quantifiable Quality of
   Service in Communication Systems at Norwegian University of Science
   and Technology - Trondheim, Norway.
*/

/*
 *  sha1.c
 *
 *  Description:
 *      This file implements the Secure Hashing Algorithm 1 as
 *      defined in FIPS PUB 180-1 published April 17, 1995.
 *
 *      The SHA-1, produces a 160-bit message digest for a given
 *      data stream.  It should take about 2**n steps to find a
 *      message with the same digest as a given message and
 *      2**(n/2) to find any two messages with the same digest,
 *      when n is the digest size in bits.  Therefore, this
 *      algorithm can serve as a means of providing a
 *      "fingerprint" for a message.
 *
 *  Portability Issues:
 *      SHA-1 is defined in terms of 32-bit "words".  This code
 *      uses <stdint.h> (included via "sha1.h" to define 32 and 8
 *      bit unsigned integer types.  If your C compiler does not
 *      support 32 bit unsigned integers, this code is not
 *      appropriate.
 *
 *  Caveats:
 *      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.
 *
 */

#include "sha1.h"

/* This is the definition of the quasigroup Q of order 16x16. */
unsigned char Q[256] ={ 10,  4,  5,  9,  6,  0, 14,  1,  2, 12,
13, 15,  3,  8, 11,  7,
 5, 11, 12,  8,  4, 14,  0,  7,  3,  2, 15, 10,  1,  9, 13,  6,
12,  5,  2, 13, 15,  8, 10, 14,  1,  3,  6,  7, 11,  0,  9,  4,
 7, 13,  3, 14,  2,  1, 11, 12,  5,  9,  4,  8,  0, 15,  6, 10,
 1,  2,  4, 10, 11,  7,  8,  9,  0, 13,  3, 14,  6, 12,  5, 15,
 4, 10,  8, 11, 13,  2, 12,  6, 14, 15,  5,  9,  7,  3,  1,  0,
 0, 14, 13,  2,  8,  3,  6,  5, 12, 11,  7,  4,  9, 10, 15,  1,
11,  6,  0,  5,  9, 13,  4,  8,  7, 10,  2,  3, 15,  1, 14, 12,
13,  8,  6,  1, 12, 10, 15,  0, 11,  5,  9,  2,  4,  7,  3, 14,
 2, 15,  1,  0,  7, 12,  5, 11,  9,  6,  8, 13, 10, 14,  4,  3,
 6, 12, 11,  7, 10, 15,  1,  3,  4,  8, 14,  0, 13,  5,  2,  9,
 8,  1, 15,  6,  3,  9,  7,  4, 10, 14, 12,  5,  2, 13,  0, 11,
15,  3,  9,  4, 14,  6,  2, 13,  8,  7,  0,  1, 12, 11, 10,  5,
14,  9,  7,  3,  1, 11, 13, 15,  6,  0, 10, 12,  5,  4,  8,  2,
 3,  0, 14, 12,  5,  4,  9, 10, 15,  1, 11,  6,  8,  2,  7, 13,
 9,  7, 10, 15,  0,  5,  3,  2, 13,  4,  1, 11, 14,  6, 12,  8
};

/* First 16 bits of a variable 'a' will be changed by 16 bits
obtained by quasigroup transformation defined below. */ #define
QFOLD(a) {\
    (a) = ((a)&0x0fffffff)|(((unsigned char)Q[( ((a)&0xf0000000)>>28)| \
    ( ((a)&0x0000f000)>> 8)])<<28);\
    (a) = ((a)&0xf0ffffff)|(((unsigned char)Q[( ((a)&0x0f000000)>>20)| \
    ( ((a)&0x00000f00)>> 8)])<<24);\
    (a) = ((a)&0xff0fffff)|(((unsigned char)Q[( ((a)&0x00f00000)>>20)| \
    ( ((a)&0x000000f0)    )])<<20);\
    (a) = ((a)&0xfff0ffff)|(((unsigned char)Q[( ((a)&0x000f0000)>>12)| \
    ( ((a)&0x0000000f)    )])<<16);\
}

/*
 *  Define the SHA1 circular left shift macro
 */
#define SHA1CircularShift(bits,word) \
                (((word) << (bits)) | ((word) >> (32-(bits))))

/* Local Function Prototyptes */ void SHA1PadMessage(SHA1Context
*); void SHA1ProcessMessageBlock(SHA1Context *);

/*
```

```
 *  SHA1Reset
 *
 *  Description:
 *      This function will initialize the SHA1Context in preparation
 *      for computing a new SHA1 message digest.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to reset.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int SHA1Reset(SHA1Context *context) {
    if (!context)
    {
        return shaNull;
    }

    context->Length_Low             = 0;
    context->Length_High            = 0;
    context->Message_Block_Index    = 0;

    context->Intermediate_Hash[0]   = 0x67452301;
    context->Intermediate_Hash[1]   = 0xEFCDAB89;
    context->Intermediate_Hash[2]   = 0x98BADCFE;
    context->Intermediate_Hash[3]   = 0x10325476;
    context->Intermediate_Hash[4]   = 0xC3D2E1F0;

    context->Computed   = 0;
    context->Corrupted  = 0;

    return shaSuccess;
}

/*
 *  SHA1Result
 *
 *  Description:
 *      This function will return the 160-bit message digest into the
 *      Message_Digest array  provided by the caller.
 *      NOTE: The first octet of hash is stored in the 0th element,
 *            the last octet of hash in the 19th element.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to use to calculate the SHA-1 hash.
 *      Message_Digest: [out]
 *          Where the digest is returned.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int SHA1Result( SHA1Context *context,
                uint8_t Message_Digest[SHA1HashSize])
{
    int i;

    if (!context || !Message_Digest)
    {
        return shaNull;
    }

    if (context->Corrupted)
    {
        return context->Corrupted;
    }

    if (!context->Computed)
    {
        SHA1PadMessage(context);
        for(i=0; i<64; ++i)
        {
            /* message may be sensitive, clear it out */
            context->Message_Block[i] = 0;
        }
        context->Length_Low = 0;    /* and clear length */
        context->Length_High = 0;
        context->Computed = 1;

    }

    for(i = 0; i < SHA1HashSize; ++i)
    {
        Message_Digest[i] = context->Intermediate_Hash[i>>2]
                            >> 8 * ( 3 - ( i & 0x03 ) );
    }

    return shaSuccess;
}

/*
 *  SHA1Input
 *
 *  Description:
 *      This function accepts an array of octets as the next portion
 *      of the message.
 *
 *  Parameters:
 *      context: [in/out]
 *          The SHA context to update
 *      message_array: [in]
 *          An array of characters representing the next portion of
 *          the message.
 *      length: [in]
 *          The length of the message in message_array
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int SHA1Input(    SHA1Context    *context,
                  const uint8_t  *message_array,
                  unsigned       length)
{
    if (!length)
    {
        return shaSuccess;
    }

    if (!context || !message_array)
    {
        return shaNull;
    }

    if (context->Computed)
    {
        context->Corrupted = shaStateError;

        return shaStateError;
    }

    if (context->Corrupted)
    {
         return context->Corrupted;
    }
    while(length-- && !context->Corrupted)
    {
    context->Message_Block[context->Message_Block_Index++] =
                    (*message_array & 0xFF);

    context->Length_Low += 8;
    if (context->Length_Low == 0)
    {
        context->Length_High++;
        if (context->Length_High == 0)
        {
            /* Message is too long */
            context->Corrupted = 1;
        }
    }

    if (context->Message_Block_Index == 64)
    {
        SHA1ProcessMessageBlock(context);
    }

    message_array++;
    }

    return shaSuccess;
}

/*
 *  SHA1ProcessMessageBlock
 *
 *  Description:
 *      This function will process the next 512 bits of the message
 *      stored in the Message_Block array.
 *
 *  Parameters:
 *      None.
 *
 *  Returns:
 *      Nothing.
 *
 *  Comments:

 *      Many of the variable names in this code, especially the
 *      single character names, were used because those were the
```

```
 *      names used in the publication.
 *
 *
 */
void SHA1ProcessMessageBlock(SHA1Context *context) {
    const uint32_t K[] =    {       /* Constants defined in SHA-1   */
                            0x5A827999,
                            0x6ED9EBA1,
                            0x8F1BBCDC,
                            0xCA62C1D6
                            };
    int           t;                /* Loop counter                */
    uint32_t      temp;             /* Temporary word value        */
    uint32_t      W[80];            /* Word sequence               */
    uint32_t      A, B, C, D, E;    /* Word buffers                */

    /*
     *  Initialize the first 16 words in the array W
     */
    for(t = 0; t < 16; t++)
    {
        W[t] = context->Message_Block[t * 4] << 24;
        W[t] |= context->Message_Block[t * 4 + 1] << 16;
        W[t] |= context->Message_Block[t * 4 + 2] << 8;
        W[t] |= context->Message_Block[t * 4 + 3];
    }

    for(t = 16; t < 80; t++)
    {
       W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
    }

    A = context->Intermediate_Hash[0];
    B = context->Intermediate_Hash[1];
    C = context->Intermediate_Hash[2];
    D = context->Intermediate_Hash[3];
    E = context->Intermediate_Hash[4];

    for(t = 0; t < 20; t++)
    {
        temp =  SHA1CircularShift(5,A) +
                ((B & C) | ((~B) & D)) + E + W[t] + K[0];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);

        B = A;
        A = temp;
        QFOLD((A));

    }

    for(t = 20; t < 40; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
        QFOLD((A));
    }

    for(t = 40; t < 60; t++)
    {
        temp = SHA1CircularShift(5,A) +
               ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
        QFOLD((A));
    }

    for(t = 60; t < 80; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
        QFOLD((A));
    }

    context->Intermediate_Hash[0] += A;
    context->Intermediate_Hash[1] += B;
    context->Intermediate_Hash[2] += C;
    context->Intermediate_Hash[3] += D;
    context->Intermediate_Hash[4] += E;

    context->Message_Block_Index = 0;
}

/*
 *  SHA1PadMessage
 *

 *  Description:
 *      According to the standard, the message must be padded to an even
 *      512 bits.  The first padding bit must be a '1'.  The last 64
 *      bits represent the length of the original message.  All bits in
 *      between should be 0.  This function will pad the message
 *      according to those rules by filling the Message_Block array
 *      accordingly.  It will also call the ProcessMessageBlock function
 *      provided appropriately.  When it returns, it can be assumed that
 *      the message digest has been computed.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to pad
 *      ProcessMessageBlock: [in]
 *          The appropriate SHA*ProcessMessageBlock function
 *  Returns:
 *      Nothing.
 *
 */

void SHA1PadMessage(SHA1Context *context) {
    /*
     *  Check to see if the current message block is too small to hold
     *  the initial padding bits and length.  If so, we will pad the
     *  block, process it, and then continue padding into a second
     *  block.
     */
    if (context->Message_Block_Index > 55)
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 64)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }

        SHA1ProcessMessageBlock(context);

        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }
    else
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 56)
        {

            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }

    /*
     *  Store the message length as the last 8 octets
     */
    context->Message_Block[56] = context->Length_High >> 24;
    context->Message_Block[57] = context->Length_High >> 16;
    context->Message_Block[58] = context->Length_High >> 8;
    context->Message_Block[59] = context->Length_High;
    context->Message_Block[60] = context->Length_Low >> 24;
    context->Message_Block[61] = context->Length_Low >> 16;
    context->Message_Block[62] = context->Length_Low >> 8;
    context->Message_Block[63] = context->Length_Low;

    SHA1ProcessMessageBlock(context);
}
```